GPU/CPU Heterogeneous Work Partitioning

Riya Savla (rds), Ridhi Surana (rsurana), Jordan Tick (jrtick)
15-740 Computer Architecture, Spring '18

ABSTRACT

With the death of Moore's Law, programmers can no longer wait for "the next latest and greatest processor" to achieve performance benefits. In an attempt to still get increasing performance, multi-core processors have been implemented. While multi-core CPUs have strong task-parallelism and direct access to disk memory, GPUs offer much more compute capabilities in data-parallelism with the caveat of a simpler, usually smaller, memory hierarchy. Thus, it is often beneficial to use heterogeneous computing -- parallel programming of algorithms that run across both CPUs and GPUs. For our project, we decided to look at heterogeneous computing to solve the unweighted Single-Source Shortest Path (SSSP) graph problem via Breadth-First Search (BFS).

To be clear, SSSP is the problem of identifying the distance from one "root" vertex in a graph to every other vertex. In solving this with BFS, we start with a "frontier" that only consists of this root vertex. To calculate the next frontier, we add all the neighbors of the current frontier's vertices that have yet to be visited. When we visit a vertex for the first time, the current iteration is the distance from the root to this vertex. Our algorithm is finished when all reachable vertices have been visited (when our next frontier is empty). Our graphs are stored in CSR format, meaning there is an array of edges and an array of offsets. This means a vertex's neighbors are stored between edges[offset[vtx]] and edges[offset[vtx+1]].

We used two papers for initial ideas and guidelines:

- Efficient Large-Scale Graph Processing on Hybrid CPU and GPU Systems - TOTEM talks about statically partitioning the workload between the CPU and the GPU and communicating the boundary edges to the other device.
- Hygraph on the other hand has a specialized data structure which enables dynamic scheduling of jobs onto both the CPU and the GPUs. This supersedes the need for static workload distribution and provides load balancing.

In analyzing this problem heterogeneously, we must take into account the strengths and weaknesses of GPUs and CPUs. This includes efficient inter-processor communication, reasonable storage of data in local memory, and maximum work parallelization. We attempt to address all of these problems, as will be discussed further on in this report.

ARCHITECTURE

We used the latedays cluster (latedays.andrew.cmu.edu) that is organized as a single head node and 17 worker nodes that process jobs submitted to a job queue and guarantees performance isolation.

Processor	**GPU: NVIDIA Tesla K40 (Kepler)[1]**	**CPU: Intel Xeon e5-2620 v3[2]**
Compute	15 SMs, 2880 CUDA cores	2 sockets, 6 cores per socket, 2-way hyperthreaded (i.e. 24 'threads')
Clock	745 Mhz Base Clock	2.4 Ghz Base Clock
Memory	12 GB RAM (288 GB/sec memory bandwidth)	16 GB RAM (59 GB/sec max memory bandwidth), 15 MB L3 Cache

1 https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf
2 https://ark.intel.com/products/83352/Intel-Xeon-Processor-E5-2620-v3-15M-Cache-2_40-GHz

STATIC PARTITIONING IMPLEMENTATION: RESULTS & DISCUSSION

Partitioning

The graphs are statically partitioned between the two processors based on a hyperparameter 'alpha' - percentage of graph **edges** given to the CPU. This is a more accurate representation of the BFS workload than using percentage of 'vertices' as we had realised by our milestone presentation.

The CPU and the GPU process their part of the 'current frontier' in parallel and synchronize at the end of every iteration i.e. they must tell each if the vertices in the frontier they just processed had outgoing edges to vertices owned by the other processor. This requires a memory transfer between the CPU and GPU in each direction over the PCI bus since discrete GPUs do not share a common address space and DRAM chip with the CPU.

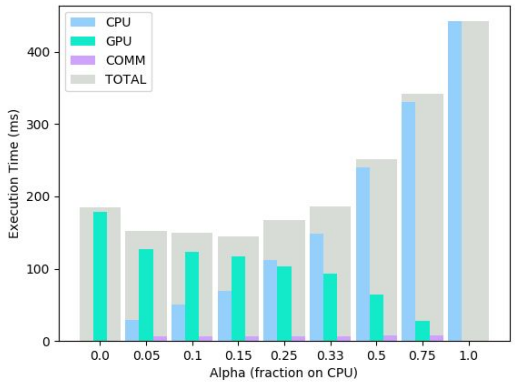
Full Frontiers

Both the CPU and GPU keep an internal boolean frontier with one bool per owned vertex (present vs. not). At the end of every iteration, each processor sends a boolean array of vertices it touched that the other processor owns. At the beginning of the next iteration, each processor syncs this 'received' frontier up with its own and proceeds.

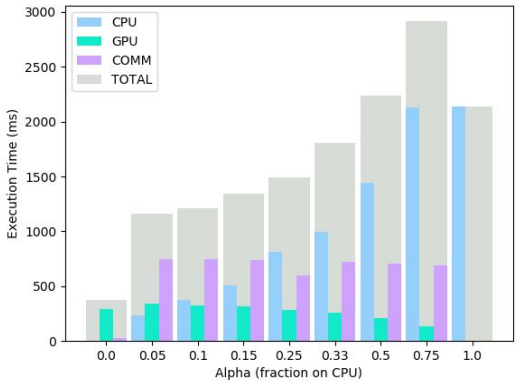
This representation requires only 2 bytes per vertex for two frontiers to be allocated on each processor, but it also requires sending that much data back and forth between them every iteration. For graphs that are partitioned such that there are very few edges that cross over the processor boundary (very likely the case for graphs with a low average degree), this turns out to be very inefficient. We have a synthetic example of this (grid1000x10000) that we show results for.

On the other hand, a full frontier also means that the CPU threads and GPU SIMD lanes can update the 'next frontier' in each iteration without the need for atomic operations. If the two vertices happen to have the same neighbour, they will both only be updating that neighbour's presence boolean value to true.

com_orkut_117m:



grid1000x1000:



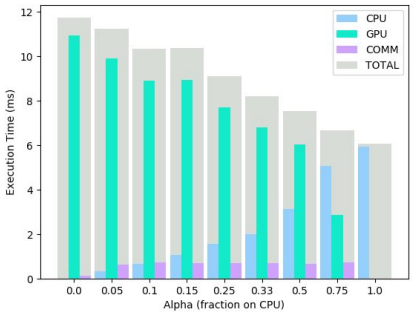
The graph on the left is a real world graph with 3m vertices, 117m edges and an average degree of 38 whereas the one on the right is a 1000 x 1000 square grid i.e. the average degree is just 4. It is very clear that the communication is a huge bottleneck.

We also tried using bit-masks as full frontiers (each bit represents one vertex). This reduced the amount of memory that had to be transferred between the processors but required an atomic operation when updating the frontiers (own and the ones to be sent over) since one integer now represented 32 vertices. Because of this tradeoff, this optimization only helped significantly for a few graphs, most notably the grid1000x1000 where communication was the clear bottleneck.

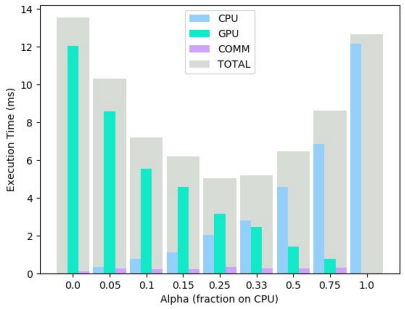
Full Frontiers with Sorted Graph Partitioning

Next we tried sorting the graph based on the degree of its vertices before partitioning it between the processors while keeping the boolean representation. We tried both orderings: CPU getting the higher degree vertices and GPU the lower, and vice-versa. CPU High performed better on most graphs than without sorting and GPU high. By assigning the CPU the higher degree vertices and having alpha determine the number of edges going to the CPU, it ends up getting significantly fewer vertices, which means that its distance array is much more likely to remain in cache and it's more likely than not that the edge array is being accessed in a more cache-friendly way. The GPU, with its high memory bandwidth and massive multithreading capability, can hide memory latency really well and doesn't do badly when assigned the lower-degree vertices. Of course, sorting does require us to re-map the final distances back in the original order, which incurs some overhead.

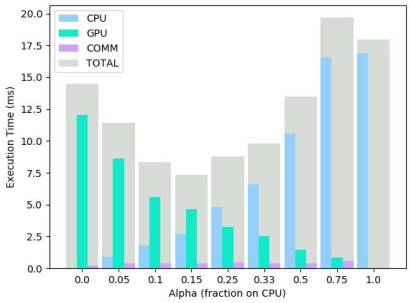
ego_twitter_2m (NOT sorted):



(sorted, CPU High):



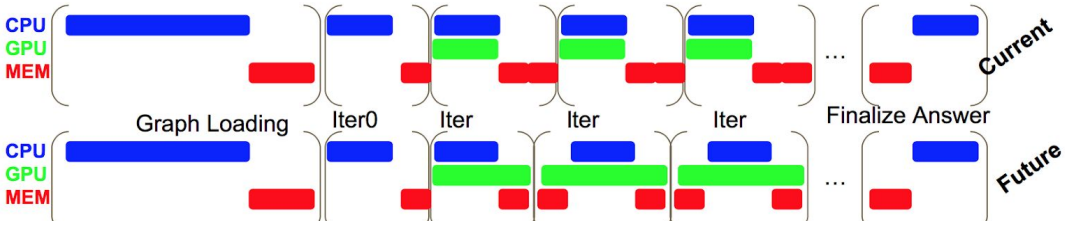
(sorted, GPU High):



On the ego_twitter_2m graph (80k vertices, 2m edges, avg. degree: 29), with sorted partitioning (CPU High) with a heterogenous approach (alpha = 0.25) performs much better than any other configuration. The GPU high version performs worse than the CPU high version.

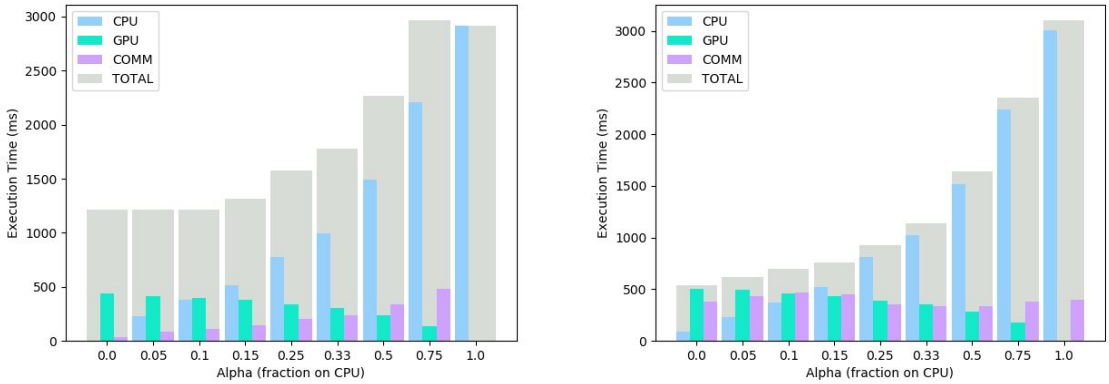
Full Frontiers with Asynchronous Memory Transfers

In the above implementations, there is an implicit barrier where each processor must wait for the other to finish its iteration before they communicate their frontiers to each other. This creates compute bubbles; ideally, we want to overlap this necessary communication with computation in order to avoid these bubbles as illustrated below:



We achieved this by using CUDA streams that allow overlapping of kernel execution with memory copies in both directions (to and from the GPU) while using cuda events for synchronization. Each operation 'queued' into a stream is guaranteed to finish before the next one begins, but operations from different streams can run concurrently. This is helpful if the GPU has a separate memory copying engine for both directions and a separate engine for launching kernels.

The GPU queues each iteration and the corresponding frontier copy to the CPU on its own stream. The CPU finishes its iteration and then queues its copy over to the GPU on a separate stream. This way, each processor is ready to begin its next iteration immediately after receiving the vertices it owns that the other processor touched in the previous iteration i.e. it doesn't have to wait for its copy to the other processor to finish.



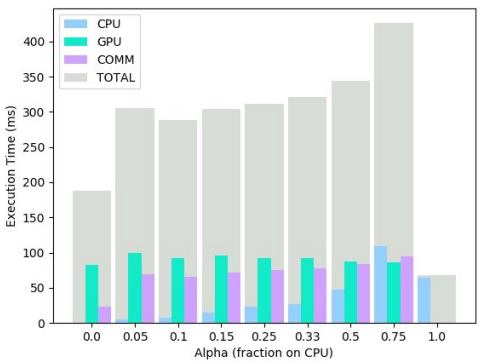
As seen above, in graphs where communication is the clear bottleneck (grid1000x1000), the asynchronous version (right) implementation performs dramatically better than the synchronous one (left), but not as much for others. [The time division for the asynchronous implementation isn't perfect because cuda event based timing when using streams is very likely to overestimate[1]]

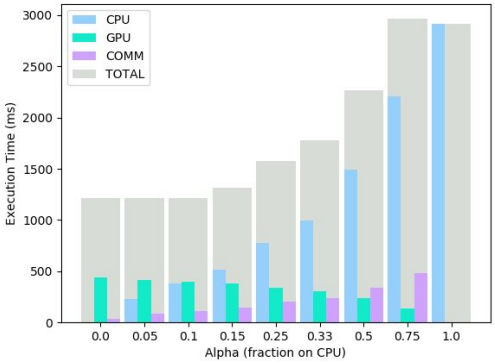
Sparse Frontiers

Originally, we represented vertex frontiers as a boolean array, where boolean[vtx] was true if vtx was in our frontier, otherwise false. This is simple to code but very memory inefficient as we must store N*sizeof(bool) data even if our frontier is only four vertices long. To be more memory efficient, we switched over to sparse frontiers, where we allocate N*sizeof(int) memory for the worst case, but now our frontier is only a list of the vertices in our current frontier. This requires additional atomic logic for modifying this array but saves memory as now if our frontier is four vertices, we can pass this frontier back and forth using only 16 bytes of memory. This is also why asynchronous memory copies are typically not useful in this scenario: the amount of logic required to setup the asynchronous transfers curbs any benefit from using them on such small memory transfers. There is an additional synchronization step required - the host i.e. CPU has to know how many vertices to copy over before it can 'queue' the copy.

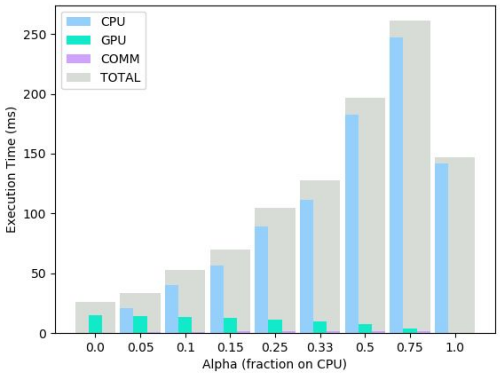
For graphs with a large number of vertices but few vertices touched every iteration, sparse frontiers have a tremendous impact. For graphs that tend to have large frontiers, it is possible that full frontiers are just more efficient if the same amount of memory is being used due to the increased number of atomic operations required by sparse frontiers. We will show examples on the next page.

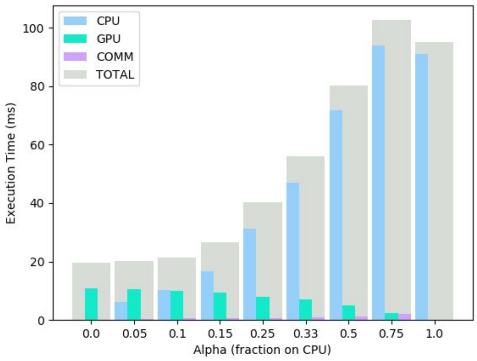
[1] https://stackoverflow.com/questions/49297293/cudaelapsedtime-with-non-default-streams





In a grid of 1000x1000 vertices of large vertex count but low degree, since frontiers are kept so sparse (less than 2000 out of 1 million), we see sparse frontiers (upper left) run 3-6 times faster than full frontiers (upper right). On the other hand, below is a random graph with 10 million edges and 1 million vertices, meaning the same vertex count but higher degree. In this case, we see full frontiers (bottom right) performing well because the frontiers are large and less synchronization is required.

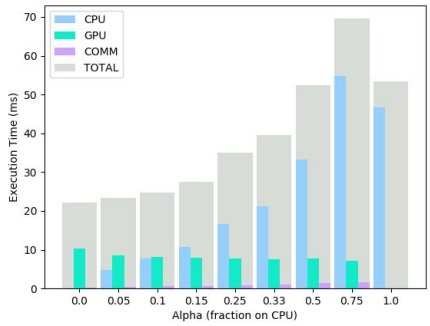


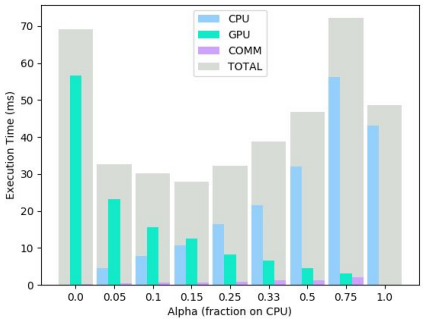


Sparse Frontiers with a single vertex assigned to a GPU warp

Another speedup we looked at relates to efficiently using the GPU's SIMD architecture: warp-based scheduling. The GPU employs both thread and data parallelism because the SM of a GPU works its own threads in a SIMD fashion, but multiple SMs can be running separate tasks in parallel. Our original naive implementation gave every thread its own vertex to process, which means each thread would be accessing a different part of edge memory. This is bad because memory stalls on a single lane of SIMD require every thread to stall, meaning every lane will stall every other lane. In an attempt to correct this, we can give each warp a vertex, instead of every thread. Now a memory stall still stalls the whole warp, but threads don't stall each other and the SM can attempt to hide this warp stalling with context switching. Also, we now have 32 threads to work on the same vertex in parallel, so we can process 32 neighbor vertices (adjacent in our edge array) in parallel which is the definition of data parallelism.

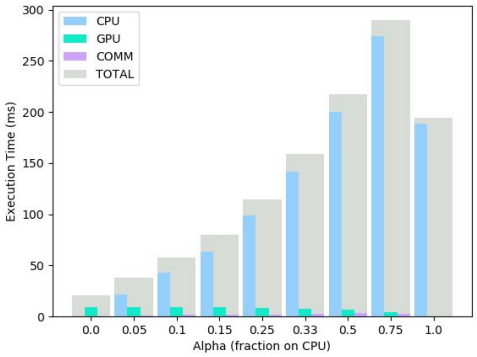
Below, we can see the results of adding warp-based processing (left) to sparse frontiers (right). The difference is not always this pronounced, but in general it is beneficial. Similar speedups are seen with full frontier implementations as well.

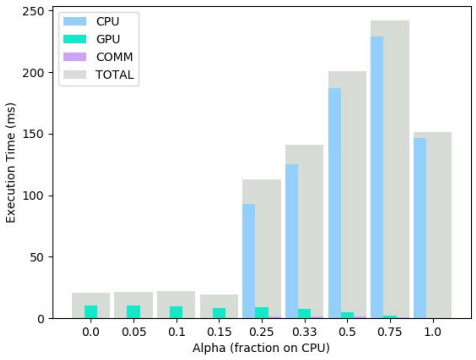




Mixed Frontiers

Next, we decided to try a boolean frontier implementation for the GPU and a sparse frontier implementation for the CPU with the communication being sparse.





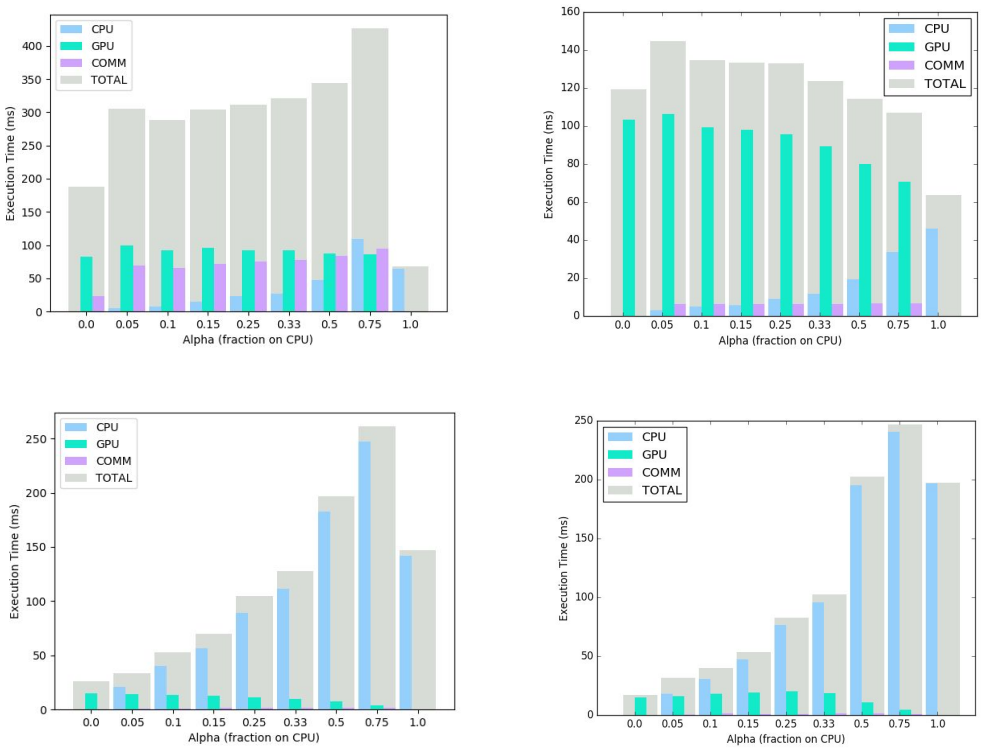
This approach performed better than individual boolean and sparse implementations for quite a few graphs, because it has the compromise of sparse communication between processors with sparse frontiers but fewer atomics during GPU computations with a full internal frontier.

Sparse Frontiers with Multiple Iterations before communicating

Memory transfers are a non-trivial part of this algorithm. Every iteration, the GPU and CPU must swap frontiers to synchronize, meaning the number of iterations our algorithm takes to find the vertex farthest away from the root determines how many memory transfers are necessary. To combat this, we can instead have the GPU and CPU sync up every "X" iterations and also mention the distance at which they saw the other processor's vertex. Now, if we had "M" memory transfers before, they have theoretically become coalesced into "M/X" transfers. Sounds good, right? For many instances it is. However, a problem develops: if the CPU was scheduled to see a vertex from the GPU at iteration "Y", it is now potentially receiving this notification at iteration "Y+X", meaning it may have incorrectly processed the distances with values longer than their shortest path. This means now we can't ignore any vertex the other processor sends us until we know that the distance they are reporting is less than our own observed distance. Furthermore, if we added distances longer than this distance in the previous X iterations, all of this work must be overwritten with smaller values.

For CPU/GPU workloads that are well separated, multiple iterations work well as redundant work is less frequently generated. For randomized or high degree graphs however, this can cause issues as the amount of redundant work becomes more detrimental than the time saved from minimizing memory transfers.

In the graphs shown below, the first row is grid 1000x1000, the second row is random with 10m edges, the first column is sparse frontiers, and the second column is sparse frontiers with syncups every four iterations. For sparse graphs like grid, the multi-iter behavior is better while for random graphs of high degree, the behavior is worse.



WHAT WE LEARNED ABOUT STATIC PARTITIONING ALGORITHMS

Overall, there are two main user choices in our static implementations: 1) which implementation to use and 2) what percent of edges to give to the CPU versus the GPU. The first is simple to understand: use async memory with large transfers, use sparse frontiers for sparse graphs, if you can afford to preprocess with a sort by degree it will be beneficial, and only run multiple iterations between syncups if you can guarantee the the CPU and GPU will rarely pass vertices to each other. In general, if you know nothing about your graph, mixed frontiers is probably the safest bet (whereby messages passed between processors are sparse but the GPU maintains an internal full frontier for minimal atomicity); the benefit of sparse frontiers is that you send N*sizeof(int) worth of messages total, whereas with full frontiers you send N*num_iterations total, so even in the worst case these transfers will be approximately equal. Full frontiers is only beneficial if your graph has high degree and/or is small enough for it not to matter.

The edge percentage is more complicated: this also depends on the CPU/GPU memory passing that results from boundary edges as well as the memory/compute capabilities of the processors. In general, we empirically determine that with powerful GPUs, it is typically a safe bet to give the GPU 75-90% of the vertices, as the GPU arch is much more data parallel and memory efficient than the CPU arch.

DYNAMIC PARTITIONING

Introduction and Motivation

The aforementioned implementations are based on static partitioning — the user pre-defines the percentage of edges to allocate to the CPU and the rest go to the GPU. Static partitioning works very well when the user has knowledge about the graph they are trying to process — i.e. information like graph size, average degree, standard deviation of degrees — and can make decisions based off these graph statistics and empirical results they see.

However, GPUs traditionally have less memory compared to CPUs and do not have disk memory support. In order to counter situations where graphs have sizes such that it's hard to predict what portion of them will fit on the GPU without hurting correctness (allocation failures) and performance, we suggest a dynamic partitioning scheme. The main idea for this scheme is that we only offload that much work to GPU that it can handle (has memory for). In this partitioning scheme, we have a current set of work that resides on the CPU and can be arbitrarily large because of the CPU disk memory. Both the CPU and GPU pick off work from this set and process the vertices in the sets they picked off. Once done, they enqueue the new set of work which can be picked by any device.

In order to be able to do this, we must ensure that once a device accesses a set of vertices it must be able to access all its edges and neighbors without having to communicate with the other device. This approach has three potential advantages: 1) the GPU is no longer prohibited by its small memory, 2) the user doesn't need to pre-process the graph at all, and 3) there exists a dynamic workload balance between both the devices. Hence, there lies an opportunity to maximize computation without one device finishing early and sitting idle as it cannot touch the vertices of another device as is possible in static implementations.

Implementations

<u>Dynamic with O(n) memory</u>
We started with a dynamic partitioning implementation that works for small- to medium-sized graphs. This allows us to compare the dynamic partitioning's pros and cons against static's pros and cons.

In this implementation, we copy the entire graph into both the CPU and the GPU such that there is no piece of work that a device cannot do. Both devices pick off work from the current frontier and process the work. Once the current work block is processed, they enqueue the new work generated from this work block onto the next frontier. They keep picking off work from the current frontier until it's empty. At this point, the current and next frontiers are switched and the processors sync their distances so they know what the other device saw and mark it as seen for themselves. There is some inherent synchronization involved in picking and enqueuing work. Only the GPU thread is responsible for switching the queues and syncing the distances for both the processors, after which it signals the CPU that it's good to go for the next iteration.
This reduces the synchronization overhead between devices at the cost of a small wait time faced by the CPU thread. This implementation only works for small- to medium-sized graphs and serves as a comparison data point.

<u>Dynamic for arbitrary sized graphs with O(1) memory</u>
Technically, we want our algorithms to be able to run graphs that have MAX_INT number of vertices. Even storing one integer for MAX_INT number of vertices is 8GB, meaning GPUs with memory less than this will fail. To design a scheme that could run on such a GPU, we need to evaluate what the GPU really needs to do meaningful work: access to edges that it can filter for the next frontier based on some sort of seen array.

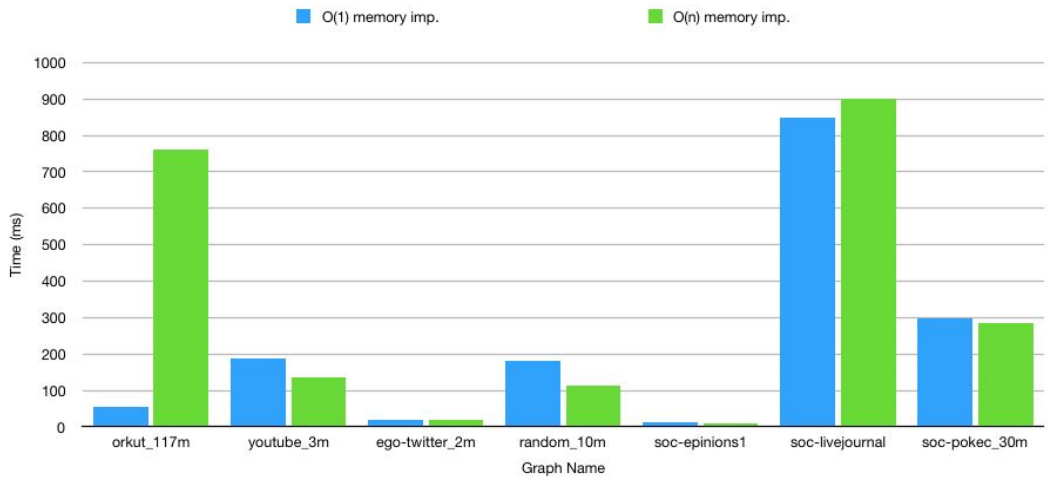
We shouldn't use a boolean seen array because this would still be 4GB, but a bitmask seen array is only 0.25GB, which is very reasonable. After this, the user can use the remainder of whatever available GPU memory exists to filter arrays of edges based on this seen array. The algorithm becomes this:

```
while(vtx exists on current frontier in cpu mem):
   Any cpu or gpu thread can grab a vertex and:
      *load its neighbors onto the cpu or gpu
      *filter these neighbors based on local cpu/gpu seen mask
      *copy the resulting unseen neighbors back to the cpu
for every vtx in next_frontier:
   cpu updates the distance array if distance[vtx]==UNSEEN
```

With this algorithm now, we can do meaningful heterogeneous computing on GPUs with just 0.3 GB of memory for ANY integer-sized graph!! This problem also naturally lends itself to multithreading: the GPU and CPU can create their own pool of worker threads that can independently grab vertices from the current frontier as long as their updates are atomic to one another. In this way, the CPU can fully utilize its multithreading design while the GPU can give each SM multiple warp's worth of edges to be SIMD-parallel but process different sets of edges on separate SMs to be task-parallel (using asynchronous CUDA streams). The workload balance between CPU and GPU is therefore determined by how many edges each processor is allowed to grab in a single chunk and how many workers each processor spawns.

Although this algorithm requires the GPU to use lots of tiny loads (fewer than 100 integers at a time based on our current empirical globals), it is surprisingly efficient because multiple SMs can be working in parallel while others wait for memory loads. On the CMU servers, this dynamic implementation performs as well as our static implementations running about 40% edges on the CPU, meaning it still has decent performance although the memory is 32x-128x less depending on which implementation runs. To be specific, the GPU requires MAX_INT/8+sizeof(int)*(2*WORK_SIZE+1)*NUM_WORKERS bytes of memory in the worst case, which is 256MB for the seen array and 256MB for the workspace if you are running 8 threads that can grab 40 edges at a time. This dynamic algorithm lends itself to nicely distributed systems and could even act towards power efficiency/regulation as smaller worker pools would not use the entire GPU.

<u>Results and Comparison</u>



We ran the above implementations on the given graphs. As can be seen, these graphs do not work quite as well as with the static partitioning. Keep in mind that these are all small and mid sized graphs and hence any benefits we get from dynamic load balancing are overshadowed by the synchronization overhead. Static partitioning on the other hand has no such overheads except communication.

CONCLUSIONS AND FUTURE DIRECTIONS

If you are going to process similar workloads or ones that change systematically, it might be worth it to run trace experiments on the available hardware and empirically pick the best strategy. On the other hand, if you have absolutely no intuition about the graph layout, dynamic partitioning might be the way to go to ensure a very even workload balance.

All of the computation shown in this report has been performed with high-end GPUs, where quite frankly it's a miracle our CPU/GPU heterogeneous computation was able to beat the GPU-only implementation at all on such small- to mid-sized graphs. This opens an interesting discussion: heterogeneous computing is a lot more feasible on mobile systems (like phones and laptops) that can only afford low- to mid- end GPUs because the CPU and GPU are more evenly matched in compute capability. In such environments, our heterogeneous system should be almost double the performance of either processor running alone. This is why **one of our future directions would be looking into integrated GPUs**, as they offer potentially more compute capability on limited systems that should try to take advantage of all their available resources. In this regard, our implementation can also naturally extend to multi-CPU and multi-GPU systems.

NOTE : We have analysed the graphs based on graph statistics and hardware resources available. However, due to the 15-213 final exams, we had no access to the GHC clusters and hence couldn't automate the process in time.

WORKS CITED

Publications:

- Efficient Large-Scale Graph Processing on Hybrid CPU and GPU Systems https://arxiv.org/pdf/1312.3018.pdf
- HyGraph: Fast Graph Processing on Hybrid CPU-GPU Platforms by Dynamic Load-Balancing http://materials.dagstuhl.de/files/17/17431/17431.AnaLuciaVarbanescu1.Preprint.pdf

Graphs:

- Stanford Large Network Dataset Collection https://snap.stanford.edu/data/

Starter Setup Code (for graph importing):

- CMU 15-418 Spring 2017 http://15418.courses.cs.cmu.edu/spring2017/article/7

APPENDIX: A BRIEF CODE REPOSITORY WALKTHROUGH

common/ : graph loading, timing code (from 15-418 starter)
cpu_kernels/ : different cpu bfs implementations (full, sparse, etc)
gpu_kernels/ : different gpu bfs implementations (full, sparse, etc)
partition/ : code for choosing how graphs are split between the cpu and gpu
hetero*/ and dynamic_partition: Each of these contains a different BFS implementation (the ones we described above + a few others we tried along the way). Each folder has a "strategies.md" for details. Each implementation generates a "bfs" executable, usually run as ./bfs <graph filename> <alpha value>.
global_config.h : ITERATIONS: number of iterations before syncing for hetero_multi
PARTITION: defines graph partitioning strategy from partition/partition.h
PLOT: define to print execution timings
DIST_CHECK: define to print only final distances out
graphs/ : a few of the graphs we work on for convenience.

run.py : python run.py -h <implementation folder name> -g <graph filename to run on, by default it runs on all the graphs from /afs/cs/academic/class/15418-s17/public/asst3_graphs> -p <optional: generate matplotlib plot of timings, PLOT must be defined in global_config.h>