An Operational Performance Model of Breadth-First Search

Sreepathi Pai
The University of Texas at Austin
sreepai@ices.utexas.edu

M. Amber Hassaan
The University of Texas at Austin
m.a.hassaan@utexas.edu

Keshav Pingali
The University of Texas at Austin
pingali@cs.utexas.edu

ABSTRACT

We introduce queueing network models for characterizing and modeling the performance of graph programs. We show that graph programs can be modelled as a queuing network whose behaviour can be mapped to a queueing network of the underlying architecture. Operational analysis of these queueing networks allows us to systematically analyze performance and identify performance bottlenecks. We demonstrate the methodology by examining a breadth-first search (BFS) implementation on both CPU and GPU.

KEYWORDS

operational analysis, queueing networks, graph algorithms, performance modelling, GPU, CPU

ACM Reference format:
Sreepathi Pai, M. Amber Hassaan, and Keshav Pingali. 2017. An Operational Performance Model of Breadth-First Search. In *Proceedings of The 1st International Workshop on Architecture for Graph Processing, Vancouver, CA, June 2017 (AGP'17),* 8 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

What limits performance of parallel graph algorithms on current processors? Folklore holds that graph algorithms are "memory-bound" and hence would benefit from large bandwidths and short memory latencies. However, recent research has found that implementations of these algorithms do not fully utilize off-chip memory bandwidth [1, 6, 7, 13]. This has been variously construed to mean that (last-level cache) locality exists in graph algorithms [1] or that high off-chip memory latency is the bottleneck [7]. Taken together, these conclusions are, of course, contradictory.

Analyzing the performance of programs that implement graph algorithms ("graph programs" for short) is tricky. Like all programs, graph program performance is an artifact of the algorithm, the implementation, the hardware as well as the actual input. Unlike most programs, however, graph program performance is significantly intertwined with "deeper" properties of the input, such as the structure of the graph.

Past studies [2, 13] have shown, for example, that graphs of road networks possess nodes with uniform degree, have a large diameter, and graph programs usually exhibit high locality when operating on them. On the other hand, graphs of social networks have small diameters (so-called *small-world* effect), have a power-law degree distribution with a few nodes having orders of magnitude more edges than others, and cause graph programs to suffer from low locality and poor load balance.

Since graphs can also be represented as matrices, these findings suggest the positions of the non-zero values and the actual values themselves in a matrix representation of a graph play an important role in performance. This is in contrast to dense parallel matrix multiplication, for example, where the size of the matrices and not the values in them play a larger role in performance. As a result, the number of factors that need to be considered to explain performance for graph programs is significantly larger, and unconsidered factors could lead to unsound conclusions.

Graph programs have been studied before [1, 3, 7, 11, 13, 14] but these studies did not produce analytical performance models. Since we intend to provide such models, our first contribution is a set of building blocks that can be used to describe graph programs as abstract machines. We call such abstract machines *Operator Machines* and we show how we can use these building blocks to construct an abstract machine for BFS.

Our second contribution is a suite of analytical performance models for implementations of these building blocks on actual hardware. These model all the factors that affect performance – the algorithm, the inputs, the actual implementation and hardware – and are based on queuing networks to allow for sound reasoning.

Finally, we validate our analytical models by examining the performance of BFS over different inputs on multiple hardware platforms.

2 QUEUEING NETWORK MODELS BACKGROUND

The performance of computer systems is often analyzed formally through the use of queuing network models [9]. In such networks, work is modeled as a *jobs* that request service at a *server*. Each server may be busy, i.e. processing one or more requests, and therefore a queue of jobs may form at a server. In general, analysis of queueing networks attempts to discover quantities like throughput of the system, utilization of each server and queue lengths at a server.

Operational analysis [5] uses operationally verifiable quantities (e.g., performance counters) to analyze queuing network models. To use operational analysis, we need to obtain at least four characteristics of the system. First, we need N, the number of jobs. Second, we need C_i, the number of jobs completing service at server i. Finally, we need $S_i(n_i)$, which is the service time per job at server i. Here, the parameter n_i represents the *load*, defined as the total number of jobs in queue and receiving service at server i.

With these characteristics and the average number of visits per job to server i (defined as $V_i = C_i/N$), we can use the following result from operational analysis [5] to determine the bottleneck device:

$$V_B S_B = \max_i V_i S_i(n_i) \quad (1)$$

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
AGP'17, Vancouver, CA
© 2017 Copyright held by the owner/author(s). 978-x-xxxx-xxxx-x/YY/MM...$15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

```
1 kernel bfs(Graph, LEVEL):
2     forall node in Graph.nodes:
3         if(node.level != LEVEL-1)
4             continue;
5
6         forall e in node.edges:
7             dst = edge_dst(e)
8             if dst.level == INF:
9               dst.level = LEVEL
```

Listing 1: Level-by-level Topology-driven BFS Pseudocode

Informally, this result formalizes the notion that a bottleneck device is one that is fully utilized and that on average takes the longest to service a job's requests. By the forced flow law, this bottleneck device determines overall throughput X_0:

$$X_0 = \frac{1}{V_B S_B} \quad (2)$$

In the following sections, we show how to map a graph program, like BFS, to a queueing network model called an *operator machine*. We then show how to analytically characterize N, C_i and n_i for an operator machine mapped to real hardware. Combining these values with S_i obtained using microbenchmarks allows us to model the performance of actual BFS implementations on real hardware.

3 THE OPERATOR MACHINE

The Operator Machine is a framework to build queueing network models of graph algorithm implementations. In this section, we illustrate how to build an operator machine for a Breadth-First Search (BFS) implementation and model its performance.

3.1 BFS Operator Machine

Listing 1 shows the pseudocode for a parallel level-by-level Breadth-First Search (BFS) kernel. Level-by-level BFS is an iterative algorithm that proceeds in rounds. In each round, the goal is to label unvisited nodes that are the children of nodes visited in the previous round. Initially, only the source node is labelled with a `level` of 0. The algorithm terminates when all nodes are labelled. Note that Listing 1 focuses on the code that is executed within each round and elides the iterative control loop that calls the kernel for each round.

Listing 1 implements what is commonly known as a topology-driven BFS [12], which visits all nodes in every BFS round. We chose this BFS not because it is the fastest – it isn't – but for simplicity of exposition. We do note that a similar variant from the Rodinia benchmark suite [4] is used extensively in architecture research as a model for irregular algorithms.

The BFS code in Listing 1 can be implemented by the abstract machine shown as a queuing network in Figure 1. To assemble this machine, we first observe that the during each round of BFS, the outermost loop (code line 2) iterates over all the nodes of the graph. The ALLNODESINPUT server performs this task in the abstract machine. Second, only nodes that were labelled in the previous round must filter through (code line 3) and this implemented by the first NODEFILTER server. Third, we must determine the edges of the nodes that filter through (code line 6), a task performed by the EDGERANGELOOKUP server. Fourth, BFS looks up the destination

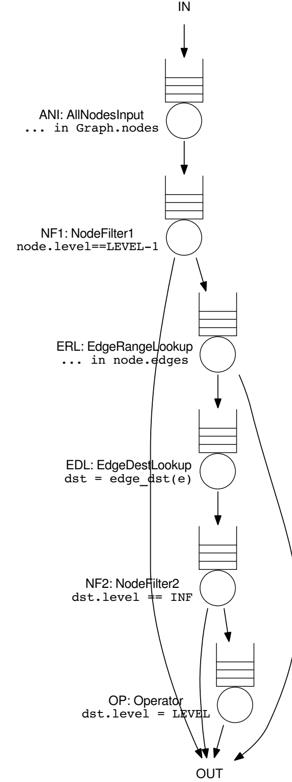

Figure 1: The Operator Machine for Level-by-level Topology-driven BFS

of each edge (code line 7), which is accomplished by the EDGEDESTLOOKUP server. Fifth, the second NODEFILTER server checks for unvisited nodes (code line 8) before finally applying the BFS operator in the OPERATOR server. Because this abstract machine exists to apply operators to graph nodes or edges, we call it an *operator machine*.

There are several advantages to representing graph programs as operator machines. Operator machines can be derived from actual implementations, as we have shown, but are a higher-level representation than code. Since each server has well-defined behaviour, it is possible to substitute different implementations for the same server. For example, EDGERANGELOOKUP could have two implementations, one for graphs laid out in memory as CSR and another for graphs laid out in COO. Finally, servers such as NODEFILTER, EDGERANGELOOKUP and EDGEDESTLOOKUP can be reused across different graph programs, forming building blocks that are largely independent of the actual graph program. This reuse is not coincidental. Our operator machine is rooted in the *operator formulation* [12] which is a framework to describe irregular algorithms of which graph algorithms form a prominent subset.

3.2 Operational Analysis of BFS

The transformation from code to operator machine is the first step in the analysis of a graph program. Once an operator machine has been built, we can analyze its performance using operational methods [5].

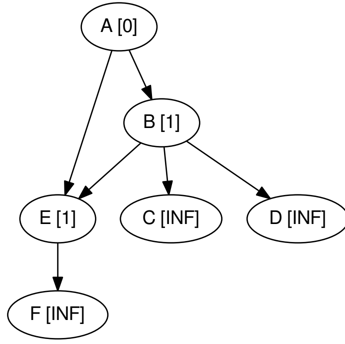

Figure 2: Input Graph for BFS. Alphabetical labels are node names, numbers in brackets are `level` values. Execution shown for `LEVEL = 2`.

To use these methods, we begin by determining the number of jobs, N, and the average number of visits per job V_i to a server i. We obtain V_i by computing C_i, the number of completions at a server i, and dividing by N. The BFS OM has six servers as shown in Figure 1.

Because we visit each node in the graph in every round, there will be as many jobs as there are vertices in the input graph. For the graph in Figure 2, N will be 6 for every round, with each job corresponding to a vertex in the graph.

Each of these jobs will visit the AllNodesInput server once, so C_{ANI} is 6. All jobs will also make one visit to the first NodeFilter server, thus C_{NF1} is also 6. Visits to EdgeRangeLookup (ERL) will change every round depending on the state of the graph. In the second round (i.e. LEVEL=2) only nodes B and E will proceed to the ERL stage, thus C_{ERL} is 2.

Since vertex E has only one edge, its job will visit EdgeDestLookup (EDL), the second NodeFilter (NF2) and Operator once. Similarly, the job corresponding to vertex B will visit EdgeDestLookup and the second NodeFilter thrice. These visits make C_{EDL} and C_{NF2} equal to 4. Since vertex E is already labelled, Operator will be visited only twice for B's edges, so C_{OP} is 3.

Using C_i and N, the average visits for job are calculated as: $V_{ANI} = 1$, $V_{NF1} = 1$, $V_{ERL} = 0.33$, $V_{EDL} = V_{NF2} = 0.66$, and $V_{OP} = 0.5$.

At this point, if we knew S_i, the service time for server i, the bottleneck server B is given by (assuming full utilization of each server and following [5]):

$$V_B S_B = max(\{V_{ANI} S_{ANI}, V_{NF1} S_{NF1}, \ldots, V_{OP} S_{OP}\}) \quad (3)$$

Assuming service time for all servers is 1 cycle, the machine will be bottlenecked for *this graph in this round* by the AllNodesInput and NodeFilter servers and the average throughput X_0 will be 1 job/cycle using Equation 2 if we ignore startup and shutdown, which we note is significant for this small graph.

The abstract BFS Operator Machine we built and analyzed in this section is most useful for designing custom graph accelerators where service times can be assumed to explore what-if scenarios. However, most graph programs today run on general purpose processors. In the following sections, we show how we can map the jobs and visits in an operator machine to actual jobs and visits on general purpose processors such as a CPU and a GPU.

Table 1: Results for the CPU

Input	Model Error	Bottleneck
RMAT	32%	L3
NY	9%	L1
FL	6%	L1

4 CPU MAPPING OF THE BFS OM

We use a single-threaded C++ implementation of topology-driven BFS. Operator Machine jobs map to memory instructions on the CPU. Correspondingly, the CPU is modelled as a closed queueing network with a three-level cache memory hierarchy – L1, L2 and L3. The completions (C_i) map to the cache hits at each level of the hierarchy and were obtained using PAPI. Service times for each level of the cache were measured using microbenchmarks.

5 CPU MODEL EVALUATION

We evaluated our model for BFS running on a Xeon 2.2GHz (Westmere) server with four 10-core processors, with each core has 32KB of L1 instruction and data caches, 256KB of L2 cache, and can run 2 SMT threads. Each of the 4 processors have 24MB of L3 cache shared among the 10 cores on the processor. We compiled BFS using gcc (6.3) with -O2 optimization level.

Microbenchmarks and the Intel Memory Checker (v3.0) tool were used to obtain service times. Latencies measured were $S_{L1} = 4$ cycles (1.33ns), $S_{L2} = 10$ cycles (4.5ns) and $S_{L3} = 46$ cycles (20.5ns). Prefetchers are active and we currently only model a MLP of 1. We currently do not model off-chip memory accesses as those "uncore" counters were unavailable on the machine we were using.

We ran BFS on a scale-free RMAT graph with 4M nodes and 32M edges, as well the road networks of NY (267K nodes, 730K edges) and FL (1M nodes, 2.7M edges).

5.1 Road Network Results

Table 1 shows the maximum relative error between the actual running time and the time obtained from the model. We observe that our model explains the performance of BFS on road networks reasonably well (less than 10% error).

5.2 RMAT Results

Figure 3 shows the actual and model time per round in cycles for RMAT graph. In early rounds, which have small amount of work, L1 is the bottleneck device, and predicts the actual time with an error less than 20%. In the middle rounds, which have large amount of work, the bottleneck device is L3, with a maximum error of 85% (round #7). The model time is lower than actual time in the middle rounds (rounds #6–9). Overall, the model correlates with actual runtime with a coefficient $r = 0.94$. This shows that modelling the memory hierarchy alone does not explain the performance of BFS entirely. Performance counter data shows that the core pipeline reports stalls due to many reasons. Mis-speculation due to branch misprediction is another factor to be considered. Therefore, we need to model the core pipeline, which we will address in future work

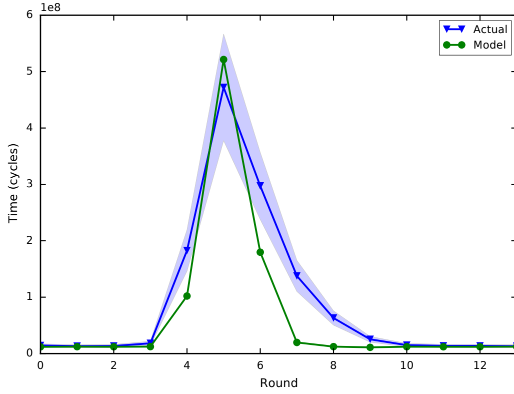

Figure 3: Operational analysis of BFS on a Xeon (Westmere), using an RMAT graph with 4M nodes, 32M edges.

```
1  kernel bfs(Graph, LEVEL):
2      for(node=tid; node < graph.num_nodes;
3                  node+=nthreads) {
4          if(level[node] != LEVEL-1)
5              continue;
6
7          r_start = graph.row_start[node];
8          r_end = graph.row_start[node + 1];
9
10         for(e = r_start; e < r_end; e++) {
11             dst = graph.edge_dst[e];
12             if(level[dst] == INF)
13               level[dst] = LEVEL;
14         }
15     }
```

Listing 2: Simplified Topology-driven BFS in CUDA

6 GPU MAPPING OF THE BFS OM

We use an implementation of BFS in CUDA running on NVIDIA GPUs to illustrate the GPU mapping of the BFS operator machine. Our description below focuses on how to model both this parallel implementation and the parallel GPU. We also show how to use data from functional simulation to identify the sources of work that cause bottlenecks.

6.1 GPU Background

A GPU is similar to a multicore CPU with each core referred to as a *streaming multiprocessor (SM)*[1]. Programs for the GPU, referred to as *kernels*, execute on the SMs using a multi-threaded single-program multiple data (SPMD) style, i.e., the code for each thread is the same. Programmers specify the number of threads to be executed when kernels are *launched* to the GPU.

On most NVIDIA GPUs, the unit of execution on an SM is not the thread, but a *warp* that comprises 32 contiguous threads which share the same program counter and thus execute the same instruction in lockstep. Since threads of the same warp can diverge, i.e. take different control paths, the GPU provides two mechanisms to allow warp divergence while maintaining lockstep execution.

Essentially, a warp executes instructions as long as one of its constituent threads is active. Inactive threads are predicated out, either by the compiler or by a dynamic mechanism known as the divergence stack [8]. The former targets small regions of code, while the latter is usually used for larger regions of code such as loops.

Up to 64 warps (i.e. 2048 threads) can be resident on an SM depending on the GPU. Functional units on the SM are shared among warps and up to 4 warps can issue instructions simultaneously. If a warp stalls, for example waiting for the result of a load, the SM switches to another warp from among the 64 warps resident on the SM. Instructions from a warp execute in order, but some GPUs also support issuing up to two instructions from the same warp.

[1] We use simplified NVIDIA terminology in this paper.

6.2 BFS on the GPU

Listing 2 is a CUDA kernel for topology-driven BFS. The number of threads executing the kernel, `nthreads`, is fixed when the kernel is launched by the programmer. Each thread also receives a unique CUDA-provided thread identifier, `tid`, which ranges from 0 to `nthreads`-1.

This code assumes the graph is stored in memory in the compressed sparse row (CSR) format. Nodes are numbered from 0 to `graph.num_nodes - 1`. The arrays `row_start` is indexed by node and contains an offset into the `edge_dst` array. Adjacent `row_start` values delimit the edges for a particular node, so the `edge_dst` values only need to contain the destination node for an edge.

The outer `for` loop maps nodes of the graph to threads in a cyclic fashion – thread *tid* will tackle node *tid*, $tid + nthreads$, $tid + 2 * nthreads$, etc. If a node was labelled in the previous iteration (Line 4), the code determines the extent of the edge list for that node (lines 7–8). An inner `for` loop then visits each edge of that node and marks any unvisited destination node of an edge as visited (lines 11–13).

6.3 Jobs

From a GPU's perspective, each dynamic instruction executed by a warp is a job that it must complete. Jobs on the operator machine therefore must be mapped to dynamic instructions on the GPU. To do this, we can use the number of jobs, the visits per job on the operator machine and the number of instructions used to implement a server on the GPU.

For example, ALLNODESINPUT is implemented by the `for` line 3 of Listing 2 which performs an assignment, a comparison, an addition and a branch. The first iteration of the loop also loads the value of `graph.num_nodes` from memory and stores it in a register. If we ignore all non-memory instructions for modelling purposes, then ALLNODESINPUT is implemented by one instruction. Now given that V_{ANI} was 1/job in the operator machine and there were $N = 5$ jobs corresponding to the 5 nodes in the operator machine, the number of dynamic instructions on the GPU is at most $N \times V_{ANI} \times 1$. This is an upper bound and not an exact count due to the warp-based execution on the GPU.

This is best illustrated by Figure 4 which shows how visits to servers on the operator machine are mapped to GPU instructions. This is a dynamic instruction trace for warp 0 obtained by running

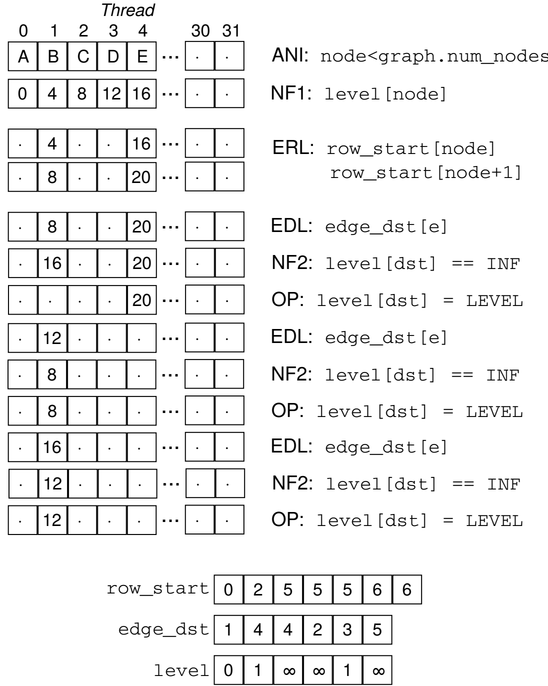

Figure 4: Dynamic memory instruction trace of warp 0 mapped to jobs. Numbers in trace represent memory offsets into arrays containing 32-bit elements. Disabled threads are marked as [·]. Array contents shown are before execution.

the code in Listing 2 on the graph in Figure 2. Each instruction is labelled with the server it implements and all non-memory instructions have been elided.

This trace shows that there is only one job for ALLNODESINPUT, i.e. $N_{ANI} = 1$ since in the CUDA implementation all nodes were mapped to threads such that all 5 nodes ended up in same warp. In general, $N_{ANI} = \lceil |V|/32 \rceil$, where V is the set of nodes in the graph. The number of NF1 jobs corresponding to the NODEFILTER is the same as N_{ANI}.

Since not every node passes through the filter, the number of EDGERANGELOOKUP jobs must be determined by counting warps that have at least one node for which EDGERANGELOOKUP is executed. In our example, two nodes B and E pass through the filter, but since they are in the same warp, the number of EDGERANGELOOKUP jobs is 2 (note that $I_{ERL} = 2$).

Warp divergence and reconvergence further complicate the counting process. The inner `for` loop in lines 10–13 implements the EDGEDESTLOOKUP, NODEFILTER and OPERATOR servers. Since each of these servers can be implemented using a single instruction, each iteration of the loop results in at least one EDGEDESTLOOKUP and NODEFILTER job. In our example, both B and E belong to the same warp but node B has degree 3, while node E has degree 1. Due to the lockstep execution of the warp, the total number of EDL jobs is 3 (not 4) since the iterations for the edges of B and E execute in the same warp. Similarly, the number of NF2 jobs is also 3. In this case, the total number of jobs resulting from the loop is given by the maximum iteration count of active threads in the warp.

Finally, conditional execution within the loop must also be taken into account when counting GPU jobs. Although B applies the operator to two edges and E to one edge, they do not do so in parallel even when they are in the same warp due to the order in which they enumerate the edges – thread 4 evaluates edge (E,F) at the same time as thread 3 looks at edge (B,E). Thus, the number of OP jobs is 3 (and not 2).

Now the total number of GPU jobs, N_G is simply the sum of all the previous calculations, i.e. $1+1+2+3+3+3 = 13$. Thus, using the number of visits to each server on the operator machine, we can obtain the number of jobs on the GPU when warp-based execution is correctly taken into account.

In practice, we can obtain GPU job counts by using instrumenting the code, through hardware performance counters for number of load (`global_load`) and store (`global_store`) instructions, or through functional simulation. In this work, we use both functional simulation and GPU counters to obtain counts of GPU jobs, and verify the former against the latter.

6.4 Visits

Since we have only considered memory instructions as jobs, the GPU is modelled by the closed queuing network model shown in Figure 5. A job can issue requests to the L2 cache or to row buffers in DRAM. For our model, we are interested primarily in V_{L2H}, V_{RBH} and V_{RBM} which count the hits in L2, hits in the row buffer and misses in row buffer respectively per job. We first describe how these quantities may be obtained operationally using counters on the GPU. Later we show that many of these quantities can also be approximated using offline simulation based on address traces.

The GPU exposes counters for total L2 transactions and total DRAM transactions. In some GPUs, DRAM accesses result in additional DRAM transactions to access ECC data, which is reflected in the total DRAM transactions but also available separately. Although GPUs do not expose them, they also contain counters to count the number of bank activations which measure the number of row buffer misses. These counters are listed in Table 2 and dividing them by the number of GPU jobs, N_G, obtained in the previous section yields the corresponding visits per job.

Functional simulation can also be used to obtain these counter values. The total number of L2 transactions can also be obtained by first counting the number of distinct 32-byte cache lines per warp for a load instruction. Summing up over all warps yields the number of L2 cache read transactions.

Since timing information is unavailable, accurate determination of L2 misses, DRAM transactions, DRAM hits and misses through functional simulation is impossible. Nevertheless, we find that simulating address traces through a fully associative cache with LRU replacement leads to reasonable approximations. Cache lines that miss in L2 may also miss in the DRAM row buffers. Again, we simulate accesses to row buffers by using a smaller fully-associative cache which uses 256-byte DRAM blocks to determine if a cache line hits or misses in the row buffer.

These simulators are necessary since the behaviour of many OM servers cannot be statically predicted. For example, the NF1 server implementation indexes the `level` array by *node*. Since adjacent threads lookup adjacent nodes, they access adjacent elements of a

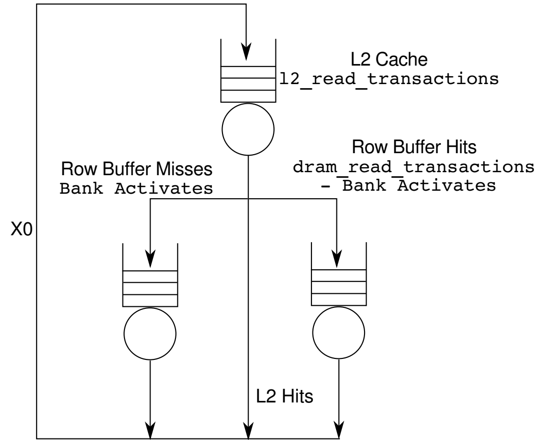

Figure 5: Closed Queuing Network Model of the GPU Memory Subsystem

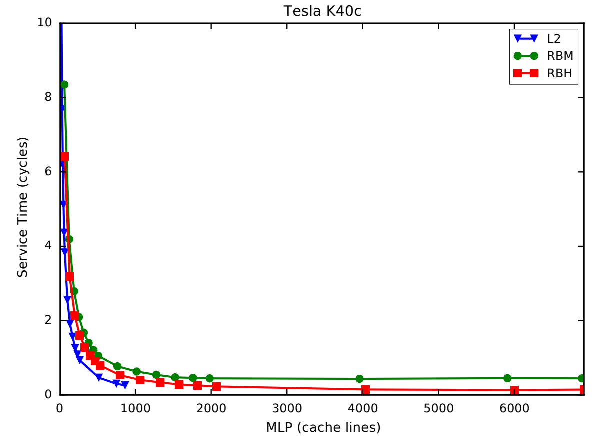

Figure 6: Memory Service Times versus load for L2 hits, row buffer hits (RBH) and row buffer misses (RBM)

Table 2: Counters and Derived Quantities. Values are for Figure 4.

Quantity	Desc.	Counter/Equation	Val.
C_{L2}	L2 cache read transactions	`l2_read_transactions`	13
C_{DRAM}	DRAM read transactions	`dram_read_transactions`	4
C_{ECC}	ECC transactions	`ecc_read_transactions`	0
C_{RBM}	Row buffer misses	Undocumented	4
C_{L2M}	L2 misses	$C_{DRAM} - C_{ECC}$	4
C_{L2H}	L2 hits	$C_{L2} - C_{L2M}$	9
C_{RBH}	Row buffer hits	$C_{DRAM} - C_{RBM}$	0

cache line or adjacent cache lines. So the total number of cache lines spanned by the warp is 4, except when the warp is not fully occupied. Accesses to the `level` array cannot be statically classified as L2 hits or misses since it is also referenced in NF2 and OP.

Accesses to `graph.edge_dst` and `level[dst]` in the inner loop (i.e. from EDL, NF2 and OP) are very different from node-indexed accesses in NF1. For these *edge-destination-induced* memory accesses, we cannot even determine the number of cache lines without functional simulation. For example, if each node had a degree of 1, then accesses to `graph.edge_dst` from within the same warp would exhibit the same behaviour as accesses to `level[node]` – they would be to adjacent memory locations. If, however, each node had a degree of 8 or more, then accesses to `graph.edge_dst` from threads within the same warp would all access different cache lines.

It is again difficult to determine the number of cache lines statically for `level[dst]` as `dst` can be almost any node in random graphs. Only by simulating the loop can the number of cache lines per warp for accesses to `graph.edge_dst` and `level[dst]` be obtained. The resulting addresses are fed to our cache simulator to estimate the number of hits and misses for both L2 and DRAM.

In contrast, the reads of `graph.row_start` in ERL are the only reference to `graph.row_start`, so we can conclude statically that the first read in any iteration will miss in the L2 cache. The second read, `row_start[node + 1]`, will nearly always hit in the cache since thread x is merely reading the same element that thread $x+1$ did in the first read of `row_start`. This temporal reuse is enabled because GPUs stall on use of a memory value, not on issue of the load, so both loads will be in flight together. Finally, since the accesses to `row_start` are to adjacent cache lines, DRAM accesses for it will mostly be row buffer hits.

For the example in Figure 4, all the arrays fit in one cache line each, so only the first access to each array will miss. We will assume all these misses will also be DRAM row buffer misses. Therefore, if `nthreads` is 32, there will be 4 misses from ANI, NF1, ERL and the first instance of EDL and all remaining accesses will be hits. We compute $V_{L2H} = 0.62$, $V_{RBH} = 0$ and $V_{RBM} = 0.31$.

For performance analysis, we only need overall counts to determine V_{L2H}, V_{RBH}, and V_{RBM}. It is technically not necessary to examine contributions from individual operator machine servers. However, as our analysis in this section shows, different operator machine servers can exhibit very different memory access behaviour even when accessing the same data region (e.g., `level` array reads in NF1 and NF2). For implementations, the operator machine therefore provides a logical framework by which contributions can be differentiated to gain better insight into the behaviour of implementations of graph algorithms.

6.5 Service Times

To complete our model, we need to know the time for a L2 hit (S_{L2H}), a row buffer hit (S_{RBH}) and a row buffer miss (S_{RBM}) on the GPU. These can be obtained using microbenchmarks accounting for the fact that multiple requests can be serviced in parallel on real hardware. Figure 6 shows that increasing memory-level parallelism (MLP) leads to decreasing service times as more requests are processed in parallel. Thus, S_{L2H}, the service time for a L2 hit is not a constant, but is a function $S_{L2H}(m)$ where m is the MLP at L2.

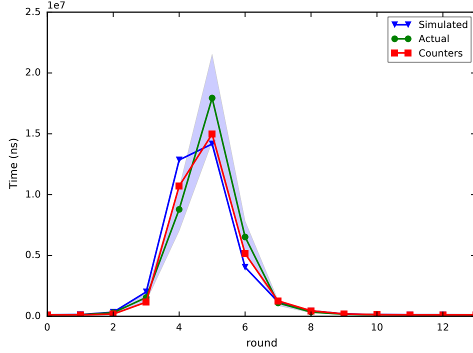

Figure 7: Model fit with characteristics derived from functional simulation and counters. Shaded area denotes ±20%.

To determine the MLP, we first determine n, the average load, which is the number of jobs in queue or receiving service. On the GPU, this is the average number of warps active per cycle, which can be measured using the achieved_occupancy counter. The MLP is then simply the product of load and the number of visits per job. For example, the number of requests in flight at L2 is $m_{L2H} = V_{L2H} \times n$.

Load can also be estimated using functional simulation. Each job on the GPU is executed by a *physical* warp. The number of physical warps is limited by the GPU and is 960 on the Tesla K40c used in this study. Assuming each job completes in 1 unit of time, the total time is W_{max} which is the maximum number of jobs executed by any physical warp. By Little's law, therefore, the estimated load is $n_{est} = N_G / W_{max}$.

Using n (or n_{est}), we can compute m_{L2H}, m_{RBH}, and m_{RBM} and use these values to obtain the average service time. At this point, we can recast equation 3 as:

$$V_B S_B = max(\{V_{L2H} S_{L2H}(m_{L2H}), V_{RBH} S_{RBH}(m_{RBH}), V_{RBM} S_{RBM}(m_{RBM})\}) \quad (4)$$

Our example graph is too small and its runtime is swamped by noise, but we will evaluate our model in the remaining sections on larger graph inputs.

7 GPU EVALUATION

We evaluate our model on a Tesla K40c GPU with BFS running on the RMAT22 graph that has 4M nodes and 32M edges. This synthetic random graph has a power-law distribution and is similar in structure to the graphs used in the Graph500 challenge. We turn ECC off since we currently cannot characterize the additional ECC DRAM transactions generated by the K40c's L2 cache.

7.1 Model Fit

Figure 7 shows the runtime obtained from our model using characteristics obtained from functional simulation (Simulation) as well from counters (Counters). In general, our model explains the performance of BFS on RMAT22 quite well. Correlation coefficients are $r = 0.95$ for simulation and $r = 0.98$ for counters. Percentage maximum error is +46% (round #4) for simulation and −38% (round #2) for counters. For simulation, the largest source of error is our estimation n_{est} which generally underestimates the load, leading to

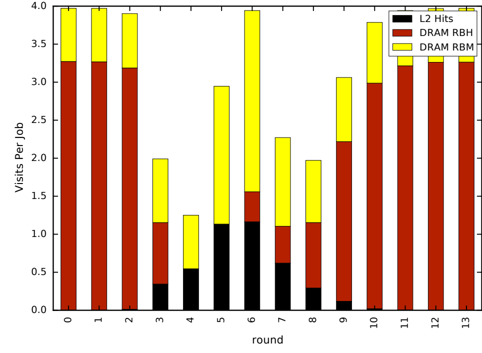

Figure 8: Visits per job (counters)

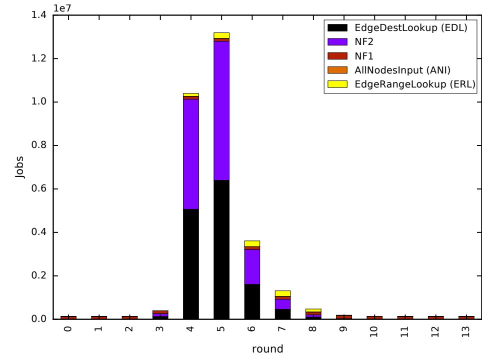

Figure 9: Jobs executed per round (functional simulation)

service time overestimates and runtimes that overshoot measured runtimes as in round #4. When this fixed by using measured load, the next source of error are the service times. Finally, we note that we do not model writes, so OP is never shown in our results. At their maximum, writes form 15% of DRAM transactions in round #5.

7.2 Visits Per Job

Using visits per job data (Figure 8) and the number of jobs per round (Figure 9), we can analyze the runtime performance of BFS on RMAT22. Note that the average number of visits per job (i.e. cache lines) never exceeds 4. Topology-driven BFS can address more than 4 cache lines up to a maximum of 32 cache lines only in in EDL and NF2. Only rounds #3–8 execute these jobs in any sufficient number, but Figure 8 shows that these rounds actually refer to less than 4 cache lines per job.

Although topology-driven BFS assigns nodes to every thread, not all nodes are active in each round. The round with the most work, round #5, has 1.5M nodes active (33%), so every third thread is idle! Further, since edges of each node are processed serially, there are very few active warps that in turn contain only a few active threads.

Bottleneck analysis shows that row buffer hits are the primary bottleneck in the initial and final rounds. In the middle rounds, however, row buffer misses dominate. From our cache simulation, we can trace these misses to the servers that generated them (Figure 10). The majority can be attributed primarily to NF2 which reads level[dst] and secondarily to EDL which obtains the value of dst from the

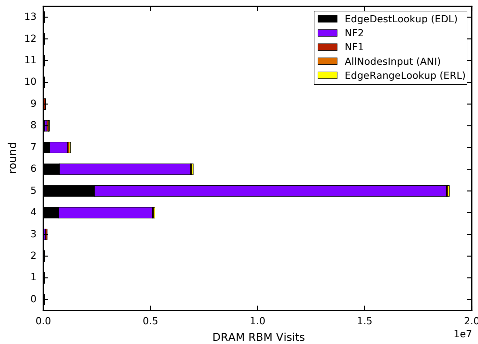

Figure 10: Contributions to DRAM RBM from OM servers

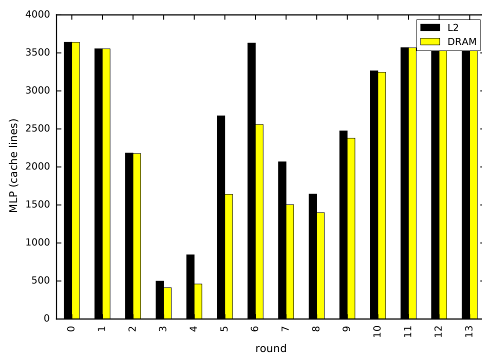

Figure 11: MLP seen at L2 and DRAM (counters)

graph structure. In contrast, `NF1`, which reads `level[node]`, is the major contributor to row buffer hits (not shown). A mere change in index changes the behaviour dramatically.

7.3 Implications

Prima facie, our results imply that topology-driven BFS can be faster if DRAM row buffer misses are sped up. However, this is only justified if the DRAM subsystem is fully loaded.

Memory bandwidth on the Tesla K40c is 288GB/s. To achieve this bandwidth at an assumed DRAM latency of 600 cycles, we need 243KB (around 7800 cache lines) in flight every cycle. Figure 11 shows that this level of memory-level parallelism is *never* achieved by topology-driven BFS in *any* round.

Because their MLP is so low, topology-driven BFS implementations (such as the BFS in Rodinia) never stress the memory system and should not be used to motivate or evaluate hardware improvements. Instead, data-driven versions of BFS [2, 10] that always assign useful work to each thread should be used. These implementations can fully load the GPU for the rounds with enough work on the RMAT22 input used in this work.

8 CONCLUSION

The Operator Machine provides a framework for modelling and interpreting the performance of graph programs. By mapping the operator machine for topology-driven BFS to the GPU, we have identified DRAM row buffer misses as the bottleneck in the rounds with most work, with row buffer hits dominating in other rounds. Further, we have shown that topology-driven implementations of BFS on the GPU have low MLP and are unable to fully load the memory subsystem. We have also shown that BFS on the CPU can be similarly modelled. Our queuing model for caches however needs to be extended to other parts of the processor pipeline to fully explain the performance of BFS on modern CPUs. In the future, we hope to model other graph programs using more sophisticated implementations on both CPUs and GPUs.

ACKNOWLEDGMENTS

The authors thank Swarnendu Biswas and the anonymous referees for comments that have improved the presentation of this work. This work was supported by the National Science Foundation under grants 1406355, 1337281, and 1618425.

REFERENCES

[1] Scott Beamer, Krste Asanovic, and David A. Patterson. 2015. Locality Exists in Graph Processing: Workload Characterization on an Ivy Bridge Server. In *IISWC 2015*. IEEE Computer Society, 56–65. DOI:http://dx.doi.org/10.1109/IISWC.2015.12

[2] Martin Burtscher, Rupesh Nasre, and Keshav Pingali. 2012. A quantitative study of irregular programs on GPUs. In *IISWC 2012*. IEEE Computer Society, 141–151. DOI:http://dx.doi.org/10.1109/IISWC.2012.6402918

[3] Shuai Che, Bradford M. Beckmann, Steven K. Reinhardt, and Kevin Skadron. 2013. Pannotia: Understanding irregular GPGPU graph applications. In *IISWC 2013*. IEEE Computer Society, 185–195. DOI:http://dx.doi.org/10.1109/IISWC.2013.6704684

[4] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC 2009*. IEEE Computer Society, 44–54. DOI:http://dx.doi.org/10.1109/IISWC.2009.5306797

[5] Peter J. Denning and Jeffrey P. Buzen. 1978. The Operational Analysis of Queueing Network Models. *ACM Comput. Surv.* 10, 3 (1978), 225–261. DOI:http://dx.doi.org/10.1145/356733.356735

[6] Assaf Eisenman, Ludmila Cherkasova, Guilherme Magalhaes, Qiong Cai, Paolo Faraboschi, and Sachin Katti. 2016. Parallel Graph Processing: Prejudice and State of the Art. In *ICPE 2016*. 85–90. DOI:http://dx.doi.org/10.1145/2851553.2851572

[7] Assaf Eisenman, Lucy Cherkasova, Guilherme Magalhaes, Qiong Cai, and Sachin Katti. 2016. Parallel Graph Processing on Modern Multi-core Servers: New Findings and Remaining Challenges. In *MASCOTS 2016*. IEEE Computer Society, 49–58. DOI:http://dx.doi.org/10.1109/MASCOTS.2016.66

[8] Wilson W. L. Fung, Ivan Sham, George L. Yuan, and Tor M. Aamodt. 2007. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *MICRO-40*. IEEE Computer Society, 407–420. DOI:http://dx.doi.org/10.1109/MICRO.2007.12

[9] Edward D. Lazowska, John Zahorjan, G. Scott Graham, and Kenneth C. Sevcik. 1984. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[10] Duane Merrill, Michael Garland, and Andrew S. Grimshaw. 2012. Scalable GPU graph traversal. In *PPOPP 2012*. ACM. DOI:http://dx.doi.org/10.1145/2145816.2145832

[11] Molly A. O'Neil and Martin Burtscher. 2014. Microarchitectural performance characterization of irregular GPU kernels. In *IISWC 2014*. IEEE Computer Society, 130–139. DOI:http://dx.doi.org/10.1109/IISWC.2014.6983052

[12] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, Muhammad Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. 2011. The tao of parallelism in algorithms. In *PLDI 2011*. ACM. DOI:http://dx.doi.org/10.1145/1993498.1993501

[13] Yuduo Wu, Yangzihao Wang, Yuechao Pan, Carl Yang, and John D. Owens. 2015. Performance Characterization of High-Level Programming Models for GPU Graph Analytics. In *IISWC 2015*. IEEE Computer Society, 66–75. DOI:http://dx.doi.org/10.1109/IISWC.2015.13

[14] Qiumin Xu, Hyeran Jeon, and Murali Annavaram. 2014. Graph processing on GPUs: Where are the bottlenecks?. In *IISWC 2014*. IEEE Computer Society, 140–149. DOI:http://dx.doi.org/10.1109/IISWC.2014.6983053