GPU Concurrency Choices in Graph Analytics

Masab Ahmad and Omer Khan
University of Connecticut, Storrs, CT, USA
{masab.ahmad, khan}@uconn.edu

***Abstract*—Graph analytics is becoming ever more ubiquitous in today's world. However, situational dynamic changes in input graphs, such as changes in traffic and weather patterns, lead to variations in concurrency. Moreover, graph algorithms are known to have data dependent loops and fine-grain synchronization that makes them hard to scale on parallel machines. Recent trends in computing indicate the rise of massively-threaded machines, such as Graphic Processing Units (GPUs). It is of paramount importance to adopt these graph algorithms efficiently on these GPU machines. However, concurrency variations are expected to play a formidable role in achieving good GPU performance. This paper performs an in-depth characterization of GPU architectural choices for graph benchmarks executing on a diverse set of input graphs. The analysis shows that performance improves by a geometric mean of 40% when optimal threads are spawned on a GPU relative to a naive choice that maximizes total thread count. Moreover, an additional 41% performance is achieved when the number of threads per GPU work group is reduced to a setting that optimizes exploitable hardware concurrency. It is also shown that algorithmic auto-tuning coupled with the right architectural choices co-optimize GPU performance.**

I. Introduction

Graph algorithms form a major subset of the overall big data analytics paradigm [1]. A variety of upcoming applications, ranging from self-driving cars to self-aware computing use graph algorithms to reach informative decisions [2]. With data becoming more and more profound, larger computing platforms such as massively parallel GPU machines are being adopted aggressively. Processors with huge memory bandwidth and vectorized processing pipelines crunch on input graph data in sensor-to-decision frameworks [3]. However, as graph algorithms exhibit irregular and unstructured behavior, communication and synchronization requirements remain high and limit performance scalability [4].

In real environments, such as road and social network graphs, dynamic changes in graph size and connectivity cause variations in inputs [5]. These effects lead to input dependence, leading to performance variations in parallel machines [6], [7]. For example, a graph having higher connectivity has a greater degree of vertex sharing, which implies higher communication requirements between processors sharing portions of the graph [8]. Therefore an input graph that exhibits higher communication requirements is expected to scale to a more limited thread count than a graph having less connectivity and vertex sharing [1]. Scalability patterns change for diverse input datasets, such as sparse versus dense, and large versus small graphs.

Performance scalability in GPUs becomes of immense importance as thousands of threads can be spawned concurrently.

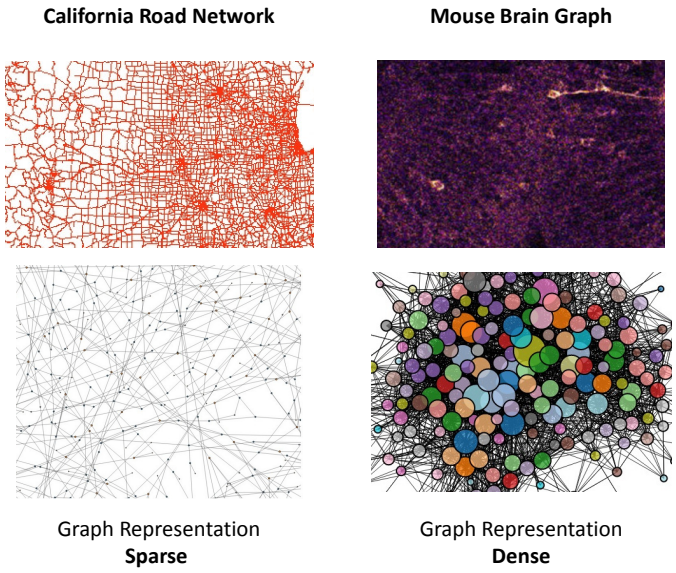

Fig. 1. Input graphs, such as California road network and mouse brain with large variations in connectivity are executed on graph algorithms. Input dependence leads to concurrency variations that must be harnessed using the right algorithmic and architectural choices.

Most graph algorithms perform shared updates using atomic instructions that induce higher synchronization costs in GPUs executing a large number of threads. Input dependent variations, amplified by these synchronization costs, cause propagative effects throughout systems with parallel machines [9]. Such input variations also result in changes to concurrency requirements, i.e., a smaller or larger number of threads need to be spawned to achieve optimal performance [10]. Moreover, GPU programming models come with the concept of warps and work groups, where a group of threads share the local memory slices. Increased vertex sharing in a graph implies that threads within a work group may bottleneck over accesses to local memory, resulting in a complex environment of architectural and concurrency variations.

The introduction of such concurrency variations lead to a slew of architectural choices, where unique inputs perform optimally at unique thread counts. This results in these different inputs scaling to different optimal architectural choices, inducing performance variability. Increasing concurrency after an optimal thread count induces synchronization that limits scalability [11]. Even more architectural choices result tradeoffs relating to data locality [12], and accesses across warps

978-1-5090-3896-1/16/$31.00 ©2016 IEEE

and work groups dictate performance. Common examples of input dependence in commodity systems include GPU setups, such as a cloud clusters [10], where users coming from different background fields submit different input types, as shown in Figure 1.

Such architectural choices are not characterized in prior works. Although some prior literature does focus on hand optimized parameters for concurrency control [13], programmers may or may not hand optimize code all the time. This results in performance variations that are characterized in this paper. Recent prior works on program auto-tuning, such as PetaBricks [14] and OpenTuner [15] optimize for a given problem where they select a target algorithm from a plethora of algorithmic choices. Such frameworks run various algorithm–input configurations based on program loops and other program parameters, and select a configuration that gives optimal performance. However, algorithmic choices are not sufficient and architectural choices still exist. Therefore, characterizing auto-tuned benchmarks for architectural concurrency choices is also necessary.

This paper makes the following contributions:

- Input-dependence in graph analytics leads to concurrency challenges in GPUs. A characterization study is presented to understand the performance implications of architectural choices in GPUs, such as selecting the right total thread count and work group size.
- The analysis of concurrency choices shows a large performance gap between naive and optimal architectural choices. Moreover, benchmarks that are auto-tuned for algorithmic choices co-optimize GPU performance with the right selection of architectural choices.

II. GPU Background

This section briefly describes the GPU programming model [16], and how to configure various architectural and concurrency control parameters.

A. GPU Programming Model

State of the art GPUs offer weak memory consistency models. Programmers have to use explicit memory fences and barriers for locally updated values to be flushed to other threads. This paper uses OpenCL [17] to program and manage GPU benchmarks. OpenCL programs use host programs written in C/C++ to launch kernels in the GPU memory space. A representative GPU architecture is shown in Figure 2.

Under OpenCL, kernels are launched with work groups (also called warps in the CUDA programming model), where a work group represents a group of threads on a vectorized GPU core sharing a local memory space. Therefore, multiple work groups are spawned on the GPU chip, with work groups sharing a global memory space, along with shader and texture memory explicitly for read-only data [18]. Atomic operations support concurrent reads and writes to global memory (mainly the L2 cache) for shared data. Further support includes local and global memory barriers, implicit thread barriers, and queue management for host to device communication and vice versa.

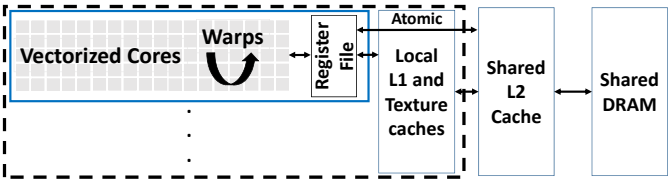

Fig. 2. Conventional GPU Processor Architecture.

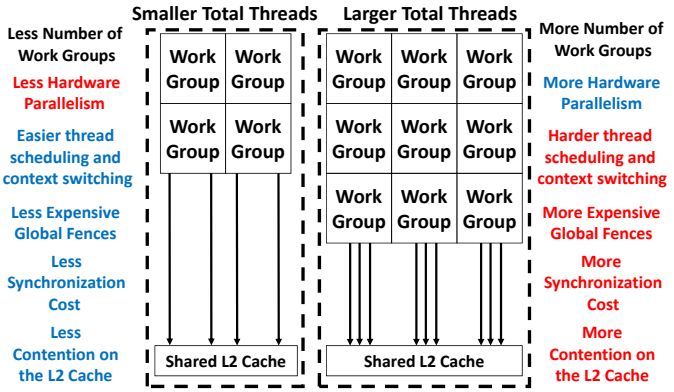

Fig. 3. Consequences of GPU threading choices on performance.

B. Thread Management in a GPU Architecture

Due to the disparity between global and local memory spaces enforced by the weaker consistency model in GPUs, programmers can control the numbers of threads spawned in a work group locally, as well as across work groups globally [19]. In modern GPUs, the work group size can be varied from 1 to 1024. The total number of threads spawned depends on the work groups used and the work group size. Again, modern GPUs can spawn up to a few million total threads that are managed by the underlying hardware schedulers. Thus, total threads improvise available parallelism and data accesses to global memory, while threads within work groups define local parallelism and data access.

III. Concurrency Variations in GPUs

This section explains how performance varies in a GPU due to variations in its concurrency controls.

A. Architectural Choices in GPU Threading

Massively-threaded GPUs can spawn thousands of threads, which are managed by an underlying hardware scheduler. This scheduler also manages thread blocks that are also known as work groups. The total number of threads running on a GPU can be calculated by multiplying the work group size with the number of work groups deployed by the programmer. Parallel programmers are generally naive and have little or no knowledge of the underlying hardware architecture, and spawn sub-optimal thread choices. This is done by keeping the work group size constant while varying the total threads, which in turn varies the number of work groups spawned on the GPU. Figure 3 shows the architectural consequences of varying thread count in a GPU setup. Smaller number of

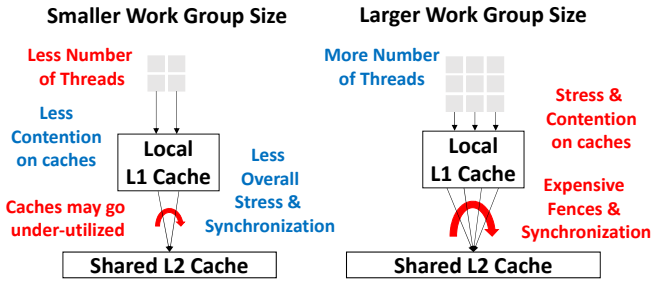

Fig. 4. Consequences of GPU work group size choices on performance.

total threads induce less exploitable hardware parallelism as less number of work groups are spawned across the chip. This reduces stress on the L2 cache as less number of work groups contend for its bandwidth. Therefore, algorithms with contended atomics and more L2 cache accesses are expected to benefit from smaller number of total threads that induce lower L2 cache contention.

A larger number of total threads allow more hardware parallelism as there are more number of work groups. However, these consequently introduce other overheads, such as harder thread scheduling, expensive global fences and synchronization via atomics, and more contention on the L2 cache. Highly concurrent algorithms that do not stress the L2 cache using synchronization and data accesses are expected to benefit from using a larger number total threads.

B. Architectural Choices in GPU Work Group Size

The variations in the work group size also lead to performance implications. A GPU work group is a collection of threads that work on a local memory chunk. A parallel programmer may keep the total number of threads spawned constant, while varying the work group size. A larger number of threads in a work group (larger work group size) results in more threads operating on the same local cache, which increases contention in data access to the L1 cache, as depicted in Figure 4. Local fences also become more expensive with larger work group sizes because more threads are required to flush data to the local cache.

A smaller work group size reduces the number of threads that share a local cache. This reduces local parallelism, while also reducing L1 cache contention and overheads associated with local fence calls. Therefore, for workloads that contend on local caches and incur overheads due to local fences, smaller work group sizes are beneficial. However, this may also cause some local cache capacity to go under-utilized, presenting potential tradeoffs in performance.

IV. Target Graph Algorithms

Graph algorithms generally consist of two main loops, an outer loop that traverses the graph vertices, and an inner loop that traverses the neighboring edges of a target vertex. Either of these loops can be parallelized in a parallel setting, with GPUs parallelizing the outer loop to take advantage of the larger available concurrent work. Most graph algorithms also go through several phases of the outer loop iterations, where phases are separated by barriers and add to the total work. Algorithmic structures are important for analyzing input sensitivity as they define the use of atomics, data access patterns, as well as thread and memory barrier requirements. Atomic primitives are generally used in inner loops, as they enable a higher degree of data sharing in graph algorithms. Barriers separate multiple inner and outer loop iterations. Therefore, the amount of exploitable parallelism not only depends on the graph algorithm but also the input graph characteristics.

A. Single Source Shortest Path (SSSP)

SSSP is a highly used workload in emerging domains, such as Unmanned Aerial Vehicles (UAVs) and self-driving cars. It involves finding a shortest path from a source vertex to all other vertices. The graph (the outer loop) is divided amongst threads, where the algorithm (Yen's optimization of the Bellman-Ford algorithm [2]), iteratively works on until the shortest path distances converge to values such that no more shorter distances/paths can be found. The inner loop updates these distances, known as relaxations, which use GPU atomics in the L2 cache to update shared edges. Variations in graph size are expected to vary available parallelism (GPU threads), while variations in the inner loop define overall communication and synchronization requirements. The version of SSSP used in this paper is taken from the Pannotia suite [20].

B. Breadth First Search (BFS)

BFS is a well known graph algorithm, and is used in conventional search, such as web crawling. It identifies parent vertices of neighboring edges, while also searching for a specific vertex in the graph in a neighbors-first manner. The outer loop is divided amongst threads, while atomic operations are used in the inner loop to update parents of shared edges. Such operations are required because multiple threads in different work groups might try to update a single shared edge. The version of BFS used in this paper is taken from the Rodinia suite [13]

C. Depth First Search (DFS)

DFS is also a well known search algorithm, which traverses vertices is a first neighbor first serve fashion. To ensure parallelism, the graph's outer loop is statically divided amongst threads, after which each thread starts a DFS on its portion of the graph. Again, atomic operations are required to update shared neighbors. This version of DFS is ported from the CRONO suite [2].

D. PageRank (PR)

PageRank is an ubiquitous algorithm that ranks web pages using a simple formula based on their connectivity and edge weights [1]. However, these are floating point calculations as PageRank values generally vary between 0 and 1. In terms of parallelization strategy, the outer loop is divided amongst threads, and the algorithm is divided into phases separated by thread barriers. The initial phases do the PageRank calculations, while the remaining phases push PageRank values to

global memory, so threads in different work groups may see updated values for subsequent phases. In some input graphs, dangling pages arise that have no outward connections to other vertices. These are catered for in the initial phase of the algorithm, where a dangling page value is computed by traversing the graph, and then later used in PageRank computations. The version of PageRank used in this paper is taken from the Pannotia suite [20]. Both PageRank, and PageRank-DP (involving dangling page calculations) are depicted as separate algorithms in this paper.

E. Triangle Counting (Tri. Cnt.)

Triangle counting is a benchmark used to identify triangles in networks. The algorithm traverses vertices, and increments counter values based on shared edges. The outer loop is divided among threads, where each thread keeps a local *triangle counter* for its portion of the graph. After a thread barrier, the threads atomically update a global triangle count value hierarchically to get the final triangle count. The triangle counting workload is ported from the CRONO suite [2].

F. Community Detection (Comm.)

Community detection identifies communities sharing a common connectivity pattern in a given graph. Due to the exact algorithm being highly serial, heuristics are applied to extract scalability at the cost of detection accuracy [3]. The algorithm runs in several phases, each of which are separated by barriers. In the initial phase, threads run over all vertices to calculate modularities, that are later used to separate vertices in respective communities in subsequent intermediate phases. Modularity computations are done via atomic operations, as threads update modularities of neighboring edges. The final phase involves a reconstruction step, which uses a heuristic to approximate communities. This workload has a much larger computation requirement, and thus scales well to high thread count with GPUs. However, due to the use of atomics and implicit thread barriers, input graph variations in density and size potentially cause concurrency variations. The community detection workload is ported from the CRONO suite [2].

G. Connected Components (CC.)

Similar in idea to community detection, the connected components algorithm identifies connectivity within a graph. The algorithm iteratively runs over the graph, and converges when no further changes in classification are possible. Input sensitivity is directly connected to performance, as convergence depends on the input graph. The parallel version of this algorithm also runs in several phases, however no shared updates are required as these updates are done locally within threads. Moreover, this benchmark has indirect memory accesses of the form *A[B[i]]*, which occur as the algorithm analyzes connected components relating vertices *A* with its neighbors *B[i]*. The connected components workload is also ported from the CRONO suite [2].

V. Methodology

A. GPU Settings

An Nvidia GTX-970 GPU is used to analyze the target graph benchamrks executing a diverse set of input graphs. The GPU has 1664 CUDA processing cores, each clocked at $1GHz$, a $4GB$ GDDR5 memory space, and memory bandwidth of 224 GB/sec. OpenCL 1.2 is used due to its diverse open source libraries and easy interface to the GPU. For characterization, thread count and work group size are varied to measure the performance sensitivity of these architectural choices. The work group size, which specifies the number of worker-threads operating on a local memory chunk, is varied from 1 to 1024, while the thread count is varied from 1 to 1024 threads per work group. The underlying GPU hardware scheduler schedules the threads on local memory.

To further compare concurrency choices across different GPUs, an Nvidia GTX 750 Ti is also utilized for characterization. This GPU has 640 CUDA cores, each clocked at $1GHz$, a $2GB$ GDDR5 memory space, and a memory bandwidth of 86.4 GB/sec. The work group size and threads per work group are varied in the same way as the GTX-970 GPU.

B. Performance Metric

For each graph benchmark, the completion time is measured for the kernel function, which is the time spent in the GPU. This work targets GPU performance characterization, hence memory transfer times between the GPU and host CPU are not considered [21]. Moreover, GPU trends indicate larger and shared memory paradigm shift that is expected to mitigate the impact of transfer times [22], [23]. The GPU completion times are acquired using the `nvprof` profiler from Nvidia.

C. Input Graphs

Several input graphs are used to identify concurrency choices in GPUs. Table I shows the datasets used in this paper. Sparse graphs, such as road networks are taken with low degrees to allow reduced communication and stress concurrency controls. Larger social networks with higher connectivity and density are taken to induce contention on the L2 cache due to the use of atomic operations. A highly dense graph modeling the mouse brain is taken to show how graph density stresses communication within work groups. A highly used Kronecker synthetic graph is also used, with it having modest density. Finally, a very large synthetic graph (Rgg) is also taken to show how concurrency is impacted when the GPU's GDDR5 memory is stressed.

D. Graph Benchmarks

All graph algorithms are taken from a variety of benchmark suites, such as CRONO [2], Pannotia [20], and Rodinia [13]. They use commonly applied vertex level parallelization strategy, and sub-optimal threading and work group sizes [30]. Sub-optimal threading corresponds to threads other than optimal, which is set to a maximum of 1024 threads per work group. Sub-optimal work group size is set to a maximum of 1024 work groups. All benchmarks use compressed sparse row (CSR) representation for input graphs and data structures.

TABLE I
INPUT DATASETS.

Input Data	# Vertices	# Edges	Avg.Deg.
Road Networks			
California (CA) [24]	1,965,206	2,766,607	1.41
Social Networks			
Facebook (FB) [25]	2,937,612	41,919,708	14.3
Livejournal (LJ) [26]	4,847,571	85,702,475	17.6
Connectomic : Brain			
Mouse Ret. 3 (CO)[27]	562	577,350	1027
Synth. Network			
Kronecker (Kron.) [28]	1,048,576	95,420,416	91
rgg-n-2-24-s0 (Rgg)[29]	16,777,216	387,553,689	23.1

E. Algorithmically Auto-tuned Graph Benchmarks

The graph benchmarks are also auto-tuned for optimal algorithmic choices using the OpenTuner framework [15]. OpenTuner is an extensible auto-tuning framework that takes in algorithms and input parameters. It learns by running various possible configurations of program loops and other algorithmic parameters, and provides an optimal configuration tuned for performance. All input graphs from Table I are input to the framework, after which OpenTuner runs multiple configurations of the benchmark–input combinations. For a more subtle analysis, it is assumed that auto-tuned benchmarks have already been optimized for total threading. The auto-tuning time is constrained to 1000 seconds, which is analogous to the studies in [15], and allows up to several thousand runs of inputs for each algorithm on the GPU. This setup is considered an optimal auto-tuner configuration in the context of learning and configuration selection.

VI. CHARACTERIZATION

A. Architectural Choices and GPU Performance

This section discusses the impact of architectural choices on raw completion times acquired from different graph algorithms and inputs, by varying work group sizes and the overall GPU threading. Table II shows the exact completion times obtained for the sub-optimal as well as the optimal choices for the GTX-970 GPU setup. The first column shows completion times obtained for sub-optimal GPU threads, while the second column shows results for sub-optimal work group sizes once optimal threads have been found. The column for optimal choice shows when both these architecture choices are optimized.

It is evident that all benchmark–input combinations improve completion time for the optimal choice. For sub-optimal choices, the maximum GPU concurrency is exploited. Although the high off-chip memory bandwidth helps overlap communication costs with computation, the unnecessary and expensive on-chip synchronization and cache contention costs add up and result in underwhelming performance. The optimal choice *systematically reduces* the number of concurrent work groups and the number of active threads per work group. In return, a balance between the amount of concurrency and the cost of communication delivers significant performance gains on the GPU. On average, the sub-optimal threading is found to be 2.63× worse, while sub-optimal work group size 1.68× worse than the completion time of the optimal choice.

TABLE II
PERFORMANCE FOR GRAPH ALGORITHMS AND INPUTS AT SUB-OPTIMAL AND OPTIMAL ARCHITECTURAL CHOICES ON THE GTX-970 GPU.

		Completion Time (ms)		
Algorithm	Input Graph	Sub-optimal Threads	Sub-optimal Work Group Size	Optimal Choice
SSSP	CA	189.4	57.22	43.31
	FB	532.1	451.3	360.2
	CO	301.4	68.23	13.42
	LJ	578.7	510.3	408.2
	Rgg.	95313	91481	82332
	Kron.	201.0	159.7	97.11
BFS	CA	77.12	25.48	16.71
	FB	71.62	32.51	21.23
	CO	82.52	40.01	19.18
	LJ	90.00	60.32	39.33
	Rgg.	182.3	175.3	125.2
	Kron.	79.01	44.37	25.81
DFS	CA	41.15	13.78	11.83
	FB	30.15	19.01	16.15
	CO	59.14	27.64	18.63
	LJ	41.13	23.51	18.40
	Rgg.	98.34	71.00	64.61
	Kron.	29.13	19.01	11.40
PR-DP	CA	411.1	291.3	278.5
	FB	400.1	305.1	267.5
	CO	397.1	299.2	270.0
	LJ	441.6	314.2	251.4
	Rgg.	561.1	422.2	373.7
	Kron.	269.2	170.1	142.3
PR	CA	56.13	40.12	32.11
	FB	71.21	53.12	31.20
	CO	71.12	41.76	33.81
	LJ	96.12	87.21	60.87
	Rgg.	341.1	298.2	272.1
	Kron.	58.12	45.14	27.13
Tri.Cnt.	CA	97.12	66.24	62.70
	FB	99.74	79.34	71.11
	CO	234.1	109.1	46.23
	LJ	164.4	129.2	66.98
	Rgg.	702.4	686.12	612.2
	Kron.	86.21	77.21	46.22
Comm.	CA	88.76	75.46	60.36
	FB	101.2	84.53	16.81
	CO	279.3	166.7	99.62
	LJ	176.2	122.1	72.72
	Rgg.	367.3	349.2	209.5
	Kron.	78.65	66.14	47.73
CC	CA	67.23	20.23	15.98
	FB	99.13	40.65	24.61
	CO	187.1	55.00	8.423
	LJ	112.4	77.23	23.13
	Rgg.	367.4	287.1	63.16
	Kron.	79.15	40.13	8.012
GeoMean		**158.7**	**101.5**	**60.36**

One notable outlier is **SSSP** running the Rgg. input graph, which takes significantly long time to complete relative to other input graphs. This happens because **SSSP** is an iterative algorithm, and the algorithm loops across the input graph multiple times to converge on the shortest path distances. For a large graph like Rgg., the shortest path algorithm takes many iterations to converge, resulting in a large completion

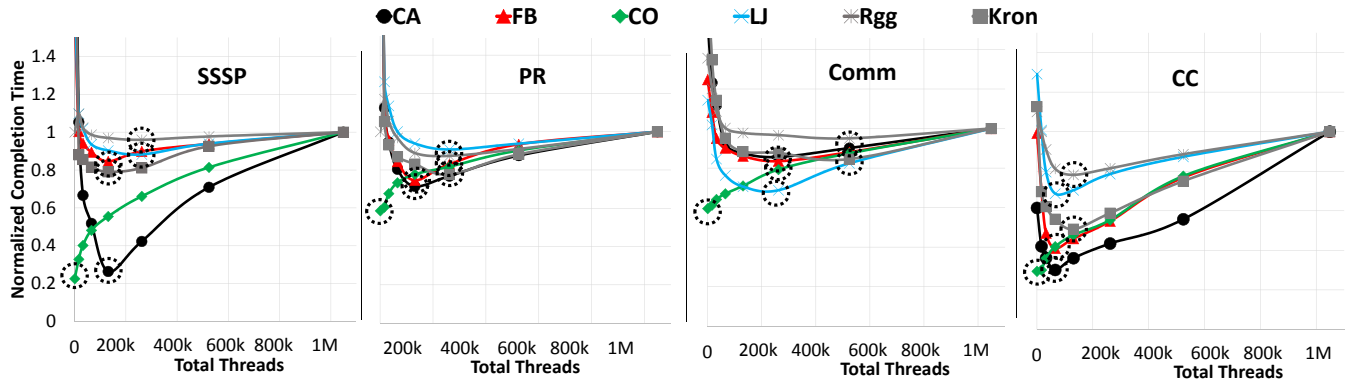

Fig. 5. Consequences of concurrency choices in GPU total threading lead to tradeoffs in parallelism and communication. The work group size is set constant at 1024, and total threads are varied. Each benchmark–input combination is normalized to the completion time of its run with maximum total threads.

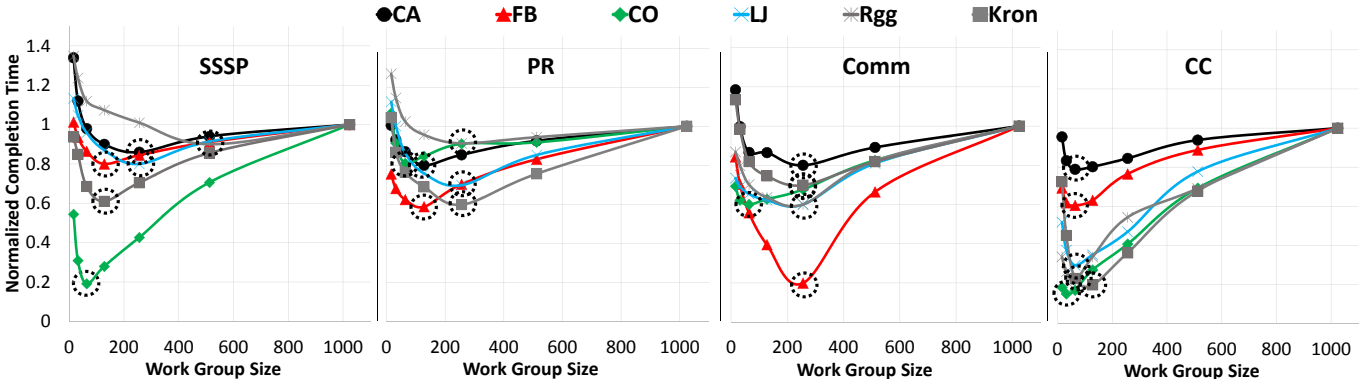

Fig. 6. Consequences of concurrency choices in GPU work group sizes lead to tradeoffs in parallelism and communication. Total threads are set constant at optimal count from Fig 5, and work group size is varied. Each benchmark–input combination is normalized to the completion time of its run with maximum work group size of 1024.

time. **BFS** and **DFS** only go through one iteration over the input graph, hence they take less time to complete relative to **SSSP**. **PR-DP** takes longer to complete than **PR**. This is because **PR-DP** has an initial phase to calculate dangling page values, which involves traversing the whole graph and then committing the values to memory. **Tri. Cnt.**, **Comm.**, and **CC** have similar completion times as **PR**, **BFS**, and **DFS** because they all go through one iteration of the input graph.

Normalized performance curves acquired by varying total threading and work group sizes are expected to show better insights with respect to the trends in concurrency variations. Figure 5 shows benchmark–input combinations normalized to the completion times acquired at the maximum total threads and maximum work group size. The performance variations are shown when the total threads spawned on the GPU are swept from 512 to 1 million, while keeping the work group size constant. Figure 6 shows benchmark–input combinations normalized to the completion times acquired at the optimal total threads from Figure 5. The performance variations are shown when the work group sizes are swept from 16 to 1024. For each sweep study, representative graph algorithms are chosen to provide insights. **SSSP** has synchronization in the inner and outer loops for each of its graph iteration. **BFS** and **DFS** have similar synchronization trends. **PR** is a highly concurrent graph algorithm. **PR-DP** and **Tri. Cnt.** have similar concurrency trends. **Comm.** goes through multiple phases, and has critical sections in the inner loop of certain phases of the algorithm. **CC** is also concurrent, although it has indirect data accesses of the form *A[B[i]]* that cause GPU caches to underperform.

For **SSSP**, most input graphs give modest performance improvements as total threads are reduced in Figure 5, with the exceptions of CA and CO. The CA graph is sparse and small compared to the rest of the graphs. Due to atomic updates in the inner loop of **SSSP**, larger total threads stress the L2 cache. Hence, reducing total threads mitigates the stress on the L2 cache, improving performance. The CO mouse brain graph has only 562 vertices, and shows best performance at 512 total threads. Adding threads beyond 512 causes some threads to remain idle and contribute only to synchronization. However, as shown in Figure 6, reducing the work group size while keeping total threads constant at 512 allows more work groups to be formed. This distribution of work among multiple work groups allows the CO graph to exploit higher level of GPU parallelism, and improve performance. Kron, which is a dense graph, also benefits from decreasing the work group size. It has significant work in the inner loop that stresses the local cache shared by the threads mapped

to each work group. Increasing the number of work groups spreads the local cache contention. Hence, Kron benefits from distributing the graph among multiple work groups. Reducing the work group size is not always beneficial, as seen for a large graph like Rgg. In the case of Rgg, a small work group size does not provide sufficient number of threads per work group to adequately overlap off-chip accesses. Hence, a relatively larger work group size works best for this graph.

PR exhibits less performance variations with changes in total threading, as depicted in Figure 5. This is because it is an inherently parallel algorithm, with little or no synchronization requirements. Moreover, it does not stress the GPU caches, thus requiring little concurrency controls. However, as shown in Figure 6, some performance variations are observed with changes in work group size. The FB graph has higher variability in the number of edges per vertex (degree). This causes some work groups to have more work than the others. Lowering the work group size reduces this high computational work by spreading it across work groups, thus mitigating load imbalance. Hence, FB performs better at low work group sizes. For same reasons as **SSSP**, the Kron graph also benefits from decreasing the work group size.

Comm. shows some variations due to total thread changes, with most input graphs requiring higher total threads than rest of the algorithms. **Comm.** has significant computation and memory accesses in its inner loops, which stresses the GPU resources. This allows higher total threading to hide latencies associated with these accesses. In terms of work group size variations, the FB graph shows great improvement in performance at lower work group sizes. This is because the variable graph structure of FB causes load imbalance, as discussed for **PR**. Reducing the work group size allows more work groups to be created, distributing work across the chip and reducing load imbalance. Moreover, allowing more work groups decreases pressure on local caches, and improves overall performance.

CC shows performance improvements at a much lower number of total threads. This happens because it has more indirect memory accesses of the form *A[B[i]* that cause contention on the L2 cache. Smaller total threads mitigate this contention by spawning a lower number of work groups across the chip. Smaller graphs (CO, CA, FB, Kron) benefit the most from mitigating contention on the L2 cache. However, larger graphs (LJ, Rgg.) need more parallelism, and hence do not benefit as much from reducing the total threads. In terms of varying the work group size, most graphs show notable improvements. CA is highly sparse and has less stress on the L2 cache, hence it benefits the least by reducing the work group size. FB still benefits from lower work group sizes by making it more load balanced. For CO, Kron., LJ, and Rgg., lower work group sizes help significantly. CO and Kron. are dense graphs that incur local cache stress amplified by indirect accesses. These bottlenecks are mitigated when multiple work groups are deployed. LJ and Rgg. are large graphs and hence benefit from distributing the work across multiple work groups.

TABLE III
PERFORMANCE FOR GRAPH ALGORITHMS AND INPUTS AT SUB-OPTIMAL AND OPTIMAL ARCHITECTURAL CHOICES ON THE GTX-750 TI GPU.

		Completion Time (ms)		
Algo.	Data	Sub-optimal Threads	Sub-optimal Work Group Size	Optimal Choice
SSSP	CA	202.4	177.3	169.2
	FB	1563	1168	1010
	CO	412.3	118.1	45.63
	LJ	990.3	930.6	723.0
	Rgg.	479813	393686	327890
	Kron.	386.21	277.1	246.2
PR	CA	160.2	135.3	113.1
	FB	159.74	99.74	91.13
	CO	111.6	95.34	64.5
	LJ	227.2	167.7	92.95
	Rgg.	2346	1703	1635
	Kron.	261.6	177.1	126.5
Comm.	CA	585.6	416.6	324.3
	FB	89.71	59.34	53.43
	CO	326.7	186.1	104.5
	LJ	322.1	205.7	134.5
	Rgg.	702.4	686.12	612.2
	Kron.	199.2	177.1	135.2
CC	CA	181.3	91.45	80.14
	FB	401.4	224.5	91.45
	CO	456.1	203.5	54.14
	LJ	601.4	437.4	300.3
	Rgg.	2135	1646	1145
	Kron.	334.6	200.4	104.1
GeoMean		**511.8**	**349.7**	**242.2**

This characterization reveals that optimizing total threads alone is not sufficient to optimize GPU performance. The right number of work groups to spawn those threads is also an important factor. Increasing the number of work groups exploits parallelism, but causes stress on the L2 cache and increases the cost of atomic operations. Lowering the number of work groups hurts parallelism, but reduces cache stress and cost of synchronization. Therefore, optimizing both total threads and work group sizes is found to be most beneficial. The characterization of parallel graph benchmarks on the GTX-970 GPU shows that performance improves by a geometric mean of 40% when total threads are optimized over a naive setting where a programmer spawns the maximum number of total threads. Moreover, when work group size is optimized in addition to total threads, GPU performance improves by an additional geometric mean of 41% over the optimal total threads setting.

B. GPU Variation

In this section a different GPU, GTX 750 Ti is considered to evaluate concurrency variations when the underlying GPU changes. As compared to GTX 970, the GTX 750 Ti has less overall CUDA cores and memory bandwidth. Table III shows the completion times acquired for the four representative graph algorithms described in Section VI-A. The Rgg graph had to be broken into two parts, and then streamed in twice as its data structures did not fit the $2GB$ memory space of the GTX 750 Ti. On average, the GTX 750 Ti performs $\sim 4\times$ worse than

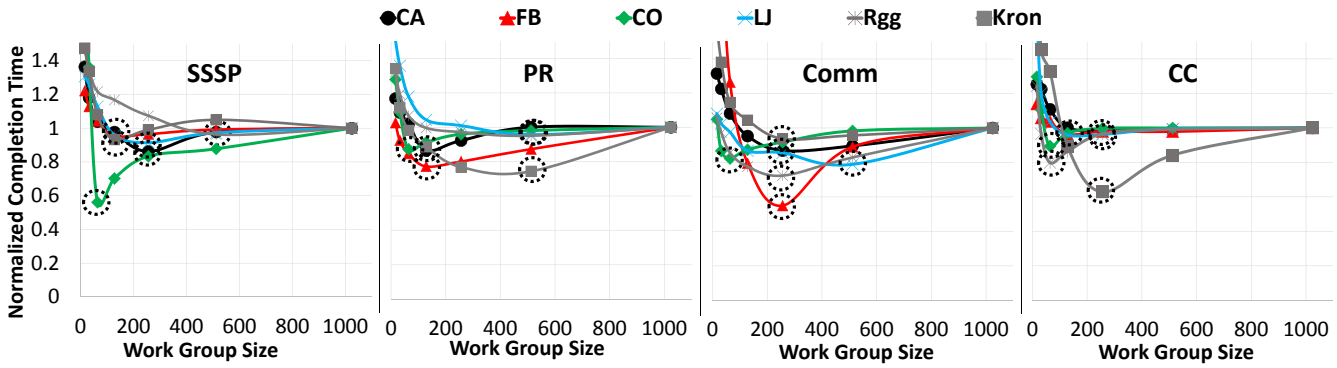

Fig. 7. Normalized performance curves for different benchmarks–graph combinations obtained using OpenTuner while varying the work group size.

the GTX-970, while the concurrency choices are still exhibited across total threads and work group sizes. Although not shown, the GTX 750 Ti shows an optimal work group size of 64, while the GTX 970 optimal work group size was seen to be between 128 and 256. This is expected as smaller GPUs have smaller compute units and local memories per work group.

On average, the sub-optimal threading is found to be 1.46× worse, while sub-optimal work group size 1.44× worse than the completion time of the optimal choice. These overheads are lower than those shown in the GTX-970, primarily because the GTX-750 Ti has less overall concurrent hardware (bandwidth, cores, caches etc.), and hence has less sensitivity to architectural choices.

C. Architectural Choices and Auto-tuned Benchmarks

In this section we evaluate OpenTuner [15] that takes in benchmark–input combinations, and optimizes for algorithmic choices. The programmer identifies loops that can be tuned, and the framework finds an optimal schedule for the loops that is tuned for performance on the target GPU. In this paper, the OpenTuner is given the optimal total threads at sub-optimal work group size, as well as the loops in the graph algorithms that can be tuned. The framework restructures and tunes for each benchmark–input combination on the GTX-970 GPU. The objective of this study is to take auto-tuned benchmarks, and study the impact of work group size architectural choice on performance.

This study is done for four representative graph algorithms (**SSSP, PR, Comm., CC**) with results shown in Table IV. OpenTuner schedules program loops optimally for each benchmark–input combination, which results in performance improvements. For all benchmarks, most input graphs show an improvement in completion time using the OpenTuner based setups over the setup without OpenTuner. On average, the sub-optimal work group size without OpenTuner is found to be 1.66× worse than the same sub-optimal setup with OpenTuner. However, the OpenTuner setup with the optimal work group size also further improves performance by an average of 14% over the sub-optimal counterpart.

Let's first compare the results of sub-optimal work group size choice with and without the OpenTuner. In **SSSP**, the CO dense graph observes a huge improvement in performance because OpenTuner better schedules data accesses and atomics in the program loops. Other dense inputs, such as Kron also observe better performance with OpenTuner. However, modest improvements are seen for relatively less dense graphs, and notably with the sparse CA input graph. **PR** does not show significant improvements with OpenTuner because its work in the program loops is dominated by floating point computations. **Comm** shows similar trends as **SSSP**, where OpenTuner performs better with graphs that perform more work in the program loops than graphs that are sparse, such as CA.

TABLE IV
PERFORMANCE RESULTS FOR ALGORITHMS USING OPENTUNER WITH SUB-OPTIMAL AND OPTIMAL ARCHITECTURAL CHOICE OF SELECTING THE WORK GROUP SIZE.

		Completion Time (ms)		
Algo.	Data	Sub-optimal Wrk. Grp. Size w/o OpenTuner	Sub-optimal Wrk. Grp. Size w/ OpenTuner	Optimal Choice with OpenTuner
SSSP	CA	57.22	49.12	42.36
	FB	451.3	373.1	356.1
	CO	68.23	22.78	12.79
	LJ	510.3	442.9	404.0
	Rgg.	91481	84135	81012
	Kron.	159.7	101.4	94.14
PR	CA	40.12	37.23	32.00
	FB	53.12	39.14	30.10
	CO	41.76	38.74	33.33
	LJ	87.21	69.07	60.1
	Rgg.	298.2	286.1	272.1
	Kron.	45.14	31.21	26.22
Comm.	CA	75.46	69.46	60.35
	FB	84.53	29.34	16.13
	CO	166.7	119.1	98.14
	LJ	122.1	91.42	72.14
	Rgg.	349.2	286.1	207.1
	Kron.	66.14	47.21	44.15
CC	CA	19.23	16.11	15.98
	FB	40.65	25.66	24.6
	CO	55.01	9.44	8.42
	LJ	77.23	24.24	23.1
	Rgg.	287.1	79.13	63.17
	Kron.	40.13	14.55	9.14
GeoMean		**123.5**	**74.30**	**64.05**

CC, which has indirect data accesses in the program loops, observes an average of 2.76× performance improvement with OpenTuner. This is again attributed to better scheduling that takes advantage of runtime information to hide the cost of indirect data accesses.

Figure 7 shows the benchmark–input combinations normalized to completion times acquired at sub-optimal work group size with OpenTuner. All benchmark-input combinations show improvements with OpenTuner, with performance curves becoming flatter compared to the non auto-tuned benchmarks in Figure 6. This happens because OpenTuner schedules optimal work among program loops to improve data access locality. Moreover, the optimal work group size generally shifts towards a larger size since OpenTuner is optimized for the maximum work group size. For example, in the case of **SSSP** the CA graph now runs optimally at a work group size of 256, which was 128 before in Figure 6. The average optimal work group size for all workloads in Figure 6 is calculated to be 192, whereas in auto-tuned workloads it is 240. However, several benchmark–input combinations still remain that improve with better selection of the work group size. Notable examples include FB for **Comm.**, Kron for **CC**, and CO for **SSSP**. This is observed because these benchmark–input combinations benefit from the higher concurrency enabled by increasing the number of work groups that OpenTuner could not automatically enable.

VII. Discussion and Future Work

Based on the characterization of graph benchmarks executing a variety of graph inputs, this work concludes that architectural choices in GPUs exist and must be exploited to improve performance. This section discusses what can and should be done to overcome the performance implications of these architectural choices. One observation is that although GPUs have extremely high memory bandwidth to off-chip memory, graph benchmarks are still constrained by on-chip resources. This is evident from the total threads and work group size sensitivity to performance, which directly ties to on-chip resource usage. Many graphs and benchmarks suffer from lack of on-chip resources when large number of total threads and work group sizes are deployed. Other big data analytic benchmarks are also expected to suffer from similar challenges. Thus, future GPU architectures can be improved with better support for on-chip caches and networks.

Another major observation shown is that input-sensitivity leads to performance sub-optimality in graph analytics. These penalties occur regardless of the GPU architecture and available parallelism. The GPU architecture and scheduler faces a gargantuan task of scheduling the optimal architecture choice for each benchmark–input combination. This challenge can be addressed by augmenting the GPU software stack with an application level scheduler, such as a recently proposed situational adaptive scheduler [31]. The scheduler can make use of the machine learning paradigm, offering sub-ms evaluation times for selecting the right architectural choices at runtime.

VIII. Related Work

Auto-tuning frameworks, such as PetaBricks [14] and OpenTuner [15] optimize for algorithmic choices. These frameworks are built to statically learn and classify algorithm–input combinations for conventional CPU processors and GPUs [10]. One shortcoming of these frameworks is that they do not characterize graph algorithms, which are known to have diverse data access patterns coupled with fine-grain synchronization. In this paper, state-of-the-art graph benchmarks are auto-tuned using the OpenTuner [15] framework. Auto-tuning program loops results in improved performance over non-auto-tuned benchmark–input combinations. However, exploiting architectural choices, such as selecting the right work group size is still found to be a beneficial choice.

Works on exploiting parallelism in GPUs also vary work group sizes at runtime to improve performance [32], [33]. Some of these works [33] show performance improvements for a few simple graph algorithms, such as BFS. However, prior works have not entirely evaluated these choices on a diverse set of input graphs. In this paper, additional architectural choices, such as total threads are considered and found to be a useful parameter for tuning performance.

Prior works in benchmarking graph algorithms are plentiful [1], such as Pannotia [20], Gunrock [21], CRONO [2], Ligra [34], and Lonestar [4]. However, none of these works characterize GPU scalability in terms of architectural choices, i.e., total threads and work group sizes. All these works also do not show input dependence and sensitivity [35]. However, some works do hint towards the fact that completion times vary with different inputs [36], [19], [31]. Such studies are very much needed if bottlenecks due to both the underlying architecture choices and benchmark–input combinations are to be characterized to improve future GPU architectures and programming models.

IX. Conclusion

This paper presents a GPU characterization study of architectural choices for graph analytics. It is seen that most concurrency bottlenecks arise due to GPU benchmarks not being tuned for each specific input in terms of total threading and warp/work group sizes. Results show that average performance overheads are 2.63× for workloads with naive concurrency controls, and 1.68× for benchmarks with optimal choice of total threads, but sub-optimal choice of work group size. Moreover, using an algorithmic auto-tuning framework improves performance, but also benefits from tuning the architectural choices as well.

Our analysis shows the following deductions:

- Input-sensitivity in graph benchmarks leads to performance bottlenecks due to under- or over-utilization of threads, as well as concurrency controls within a GPU.
- The right selection of these architectural choices leads to a significant improvement in GPU performance. Furthermore, algorithmic auto-tuning coupled with architectural choices co-optimizes GPU performance.

ACKNOWLEDGMENT

This work was supported in part by Semiconductor Research Corporation (SRC). This work was also funded in part by the U.S. Government under a grant by the Naval Research Laboratory, and a Department of Education GAANN Fellowship. We would also like to thank the anonymous reviewers for their constructive feedback.

REFERENCES

[1] N. Satish, N. Sundaram, M. M. A. Patwary, J. Seo, J. Park, M. A. Hassaan, S. Sengupta, Z. Yin, and P. Dubey, "Navigating the maze of graph analytics frameworks using massive graph datasets," in *Proc. of the 2014 ACM SIG. Int. Conf. on Management of Data (SIGMOD)*. NY, USA: ACM, 2014.

[2] M. Ahmad, F. Hijaz, Q. Shi, and O. Khan, "Crono : A benchmark suite for multithreaded graph algorithms executing on futuristic multicores," in *Proc. of IEEE Int. Symposium on Workload Characterization*, ser. IISWC, 2015.

[3] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed graphlab: A framework for machine learning and data mining in the cloud," *Proc. VLDB Endow.*, vol. 5, no. 8, pp. 716–727, Apr. 2012.

[4] M. Burtscher, R. Nasre, and K. Pingali, "A quantitative study of irregular programs on gpus," in *Workload Characterization (IISWC), 2012 IEEE International Symposium on*, Nov 2012, pp. 141–151.

[5] J. Mondal and A. Deshpande, "Managing large dynamic graphs efficiently," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '12. New York, NY, USA: ACM, 2012, pp. 145–156.

[6] Y. Ding, J. Ansel, K. Veeramachaneni, X. Shen, U.-M. O'Reilly, and S. Amarasinghe, "Autotuning algorithmic choice for input sensitivity," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2015. New York, NY, USA: ACM, 2015, pp. 379–390.

[7] M. Ahmad, K. Lakhsminarasimhan, and O. Khan, "Efficient parallelization of path planning workload on single-chip shared-memory multicores," in *Proceedings of the IEEE High Performance Extreme Computing Conferenc*, ser. HPEC '15. IEEE, 2015.

[8] P. Harish, V. Vineet, and P. J. Narayanan, "Large graph algorithms for massively multithreaded architectures," *Technical Report Number IIIT/TR/2009/74*, 2009.

[9] A. Davidson, S. Baxter, M. Garland, and J. D. Owens, "Work-efficient parallel GPU methods for single source shortest paths," in *Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium*, May 2014, pp. 349–359.

[10] P. M. Phothilimthana, J. Ansel, J. Ragan-Kelley, and S. Amarasinghe, "Portable performance on heterogeneous architectures," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13. New York, NY, USA: ACM, 2013, pp. 431–444.

[11] V. Prabhakaran, M. Wu, X. Weng, F. McSherry, L. Zhou, and M. Haridasan, "Managing large graphs on multi-cores with graph awareness," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 4–4.

[12] G. Kurian, O. Khan, and S. Devadas, "The locality-aware adaptive cache coherence protocol," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY, USA: ACM, 2013, pp. 523–534.

[13] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, Oct 2009, pp. 44–54.

[14] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe, "Petabricks: A language and compiler for algorithmic choice," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '09. New York, NY, USA: ACM, 2009, pp. 38–49.

[15] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe, "Opentuner: An extensible framework for program autotuning," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, ser. PACT '14. New York, NY, USA: ACM, 2014, pp. 303–316.

[16] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Prof., 2010.

[17] B. Gaster, L. Howes, D. R. Kaeli, P. Mistry, and D. Schaa, *Heterogeneous Computing with OpenCL*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

[18] R. Kaleem, R. Barik, T. Shpeisman, B. T. Lewis, C. Hu, and K. Pingali, "Adaptive heterogeneous scheduling for integrated gpus," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, ser. PACT '14. NY, USA: ACM, 2014, pp. 151–162.

[19] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic warp formation and scheduling for efficient gpu control flow," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 40. Washington, DC, USA: IEEE Computer Society, 2007, pp. 407–420.

[20] S. Che, B. Beckmann, S. Reinhardt, and K. Skadron, "Pannotia: Understanding irregular gpgpu graph applications," in *IEEE Int. Symposium on Workload Characterization (IISWC)*, Sept 2013, pp. 185–195.

[21] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the gpu," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP 2015. New York, NY, USA: ACM, 2015, pp. 265–266.

[22] W. Li, G. Jin, X. Cui, and S. See, "An evaluation of unified memory technology on nvidia gpus," in *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on*, May 2015, pp. 1092–1098.

[23] K. Koukos, A. Ros, E. Hagersten, and S. Kaxiras, "Building heterogeneous unified virtual memories (uvms) without the overhead," *ACM Trans. Archit. Code Optim.*, vol. 13, no. 1, pp. 1:1–1:22, Mar. 2016.

[24] J. Leskovec and et al, "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters," 2008.

[25] J. Mcauley and J. Leskovec, "Discovering social circles in ego networks," *ACM Trans. Knowl. Discov. Data*, vol. 8, no. 1, pp. 4:1–4:28, Feb. 2014.

[26] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, Dec. 2011.

[27] J. W. Lichtman, H. Pfister, and N. Shavit, "The big data challenges of connectomics," in *Nature Neuroscience 17*, Sept 2014.

[28] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani, "Kronecker graphs: An approach to modeling networks," *J. Mach. Learn. Res.*, vol. 11, pp. 985–1042, Mar. 2010.

[29] R. A. Rossi and N. K. Ahmed, "rgg-n-2-20-s0 - dimacs10," 2013. [Online]. Available: http://networkrepository.com/rgg_n_2_20_s0.php

[30] Q. Xu, H. Jeon, and M. Annavaram, "Graph processing on gpus: Where are the bottlenecks?" in *Workload Characterization (IISWC), 2014 IEEE International Symposium on*, Oct 2014, pp. 140–149.

[31] M. Ahmad, C. J. Michael, and O. Khan, "A case for a situationally adaptive many-core execution model for cognitive computing workloads," *ASPLOS 2016 Int. Workshop on Cognitive Arch., (CogArch)*, 2016.

[32] H. Wu and et. al., "Compiler-assisted workload consolidation for efficient dynamic parallelism on GPU," *CoRR*, vol. abs/1606.08150, 2016.

[33] J. Wang, N. Rubin, A. Sidelnik, and S. Yalamanchili, "Dynamic thread block launch: A lightweight execution mechanism to support irregular applications on gpus," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ser. ISCA '15. New York, NY, USA: ACM, 2015, pp. 528–540.

[34] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '13. New York, NY, USA: ACM, 2013, pp. 135–146.

[35] S. K. Haider, W. Hasenplaugh, and D. Alistarh, "Lease/release: Architectural support for scaling contended data structures," in *Proceedings of the 21st ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, ser. PPoPP '16. NY, USA: ACM, 2016, pp. 1–12.

[36] Y. Wu, Y. Wang, Y. Pan, C. Yang, and J. D. Owens, "Performance characterization of high-level programming models for gpu graph analytics," in *Workload Characterization (IISWC), 2015 IEEE International Symposium on*, Oct 2015, pp. 66–75.