Dynamic Parallelism for Simple and Efficient GPU Graph Algorithms

Peter Zhang¹
pazhang@indiana.edu

Eric Holk¹
eholk@indiana.edu

John Matty²
jcmatty@sei.cmu.edu

Samantha Misurda²
slmisurda@sei.cmu.edu

Marcin Zalewski¹
zalewski@indiana.edu

Jonathan Chu²
jchu@sei.cmu.edu

Scott McMillan²
smcmillan@sei.cmu.edu

Andrew Lumsdaine¹
lums@indiana.edu

¹Center for Research in Extreme Scale Technologies (CREST) ²Software Engineering Institute (SEI)
Indiana University Carnegie Mellon University
Bloomington, IN Pittsburgh, PA

ABSTRACT

Dynamic parallelism allows GPU kernels to launch additional kernels at runtime directly from the GPU. In this paper we show that dynamic parallelism enables relatively simple high-performance graph algorithms for GPUs. We present breadth-first search (BFS) and single-source shortest paths (SSSP) algorithms that use dynamic parallelism to adapt to the irregular and data-driven nature of these problems. Our approach results in simple code that closely follows the high-level description of the algorithms but yields performance competitive with the current state of the art.

1. INTRODUCTION

Although graphics processing units (GPUs) are best known for their impressive floating point performance, their high memory bandwidth means GPUs can also accelerate memory-bound applications such as graph algorithms. The irregular nature of graph algorithms, however, makes it difficult to effectively utilize parallel hardware. NVIDIA's Kepler-class and newer GPUs support *dynamic parallelism* (DP), which allows GPU kernels to launch additional kernels from the GPU in order to accommodate parallelism discovered at runtime without CPU. To date, DP has been largely under-utilized in the literature. Instead, efficient GPU graph algorithms are often implemented using multiple kernels, with each specialized for certain degrees or forms of parallelism (e.g., [4] [6]). There is a disadvantage to needing so much specialization, however, as it increases the complexity of the algorithm.

Using breadth-first search (BFS) as an illustrative example, we show that DP is well-suited to graph algorithms, as the resulting implementation of the BFS algorithm closely follows the abstract algorithm definition. The basic algorithm does

not perform as well as the current state of the art, but by applying a small set of modular improvements (which expose additional aspects and advantages of DP), we show that we can improve our algorithm's performance to be competitive with the current state of the art.

Specifically, our paper makes the following contributions:

- We demonstrate the applicability of DP to graph algorithms through a novel implementation of breadth-first search, which is significantly simpler in its expression than existing GPU graph algorithms (Section 3).
- We identify performance bottlenecks in our naive DP implementation and show that only minor modifications are necessary to overcome these bottlenecks (Section 3.1).
- We show that DP is generally applicable to graph algorithms through an implementation of single source shortest paths (SSSP) (Section 4).

2. BACKGROUND

2.1 Graph 500

The Graph 500 benchmark [7] is the leading graph benchmark in HPC, measuring the performance of breadth-first search, and the SSSP kernel is in development. Graph 500 utilizes a performance measure of Traversed Edges Per-Second (TEPS), which is calculated as follows:

$$TEPS = \frac{EdgesTraversed}{TotalTraversalTime}$$

The performance of benchmark implementations are usually reported using *GTEPS*, where 1 *GTEPS* = 10^9 *TEPS*.

2.2 CUDA and Dynamic Parallelism

CUDA programs feature a number of kernels that run on the GPU with a large number of threads. These threads are divided into a grid of blocks and within a block threads are further grouped into warps. Before dynamic parallelism was introduced, CUDA required the kernels to be launched by the host program, which can lead to underutilizing the GPU in inherently dynamic algorithms like graph problems. *Dynamic parallelism* allows the launch of child kernels directly from within a kernel [1]. For our purposes, this allows a kernel to spawn a new kernel to process a vertex's adjacent vertices without incurring a round trip to the CPU.

Permission to make digital or hard copies of all or part 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 components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

IA '3 2015, November 15-20, 2015 Austin, TX, USA

Copyright 2015 ACM ISBN 978-1-4503-4001-4/15/11 ...\$15.00.

DOI: <http://dx.doi.org/10.1145/2833179.2833189>

Variant	Scale 15		Scale 20		Scale 24	
Base	0.017		0.050		0.060	
W. Coop.	0.142	754%	0.854	1600%	0.707	1071%
Edge Sort.	0.157	11%	0.925	8%	0.725	3%

Table 1: The effect of various changes to our algorithm (GTEPS mean of 10 runs on a Graph500 graph and change relative to the previous row).

3. BFS ALGORITHM DESCRIPTION

We use a *level-synchronized* algorithm for BFS. The algorithm has two loops: the outer loop iterates over frontier queues and the inner loop iterates over each of a vertex’s outgoing edges, adding the edges’ targets to the next iteration’s frontier queue. Both the inner and the outer loops can be executed in parallel, which are represented by `kernel1` and `kernel2` in our algorithm, respectively. `kernel1` performs one level of the search, calling `kernel2` which processes one vertex from the current queue, adds each of its adjacent vertices to the queue for the next iteration, and sets the parent of the adjacent vertices to the current vertex in parallel. This is the essence of our dynamic parallelism based parallel breadth-first search algorithm (BFS-DP). The basic form of our algorithm is implemented in a mere 29 non-empty lines of code.

Kernels `kernel1` and `kernel2` make use of two additional data structures: the parent map and the queue. The parent map is an array with one entry per vertex that stores the parent discovered for each vertex. There are several choices for how to represent the queue, such as a dynamically sized queue, a fixed-size boolean-vector or a fixed-size bitmap. We chose a bitmap for its simplicity. The bitmap has one bit per vertex, and each vertex’s corresponding bit is set to true if that vertex is in the current frontier.

This algorithm is very straightforward and also quite readable but does not perform well (0.06 GTEPS on scale 24) since launching kernels for nodes with only a few adjacent vertices hurts performance.

3.1 Improvements

Due to the simplicity of our code, our algorithm can be extended with a number of modular improvements to change the performance. Here we discuss several changes that improved performance. Later on in Section 3.2 we will describe attempted improvements that actually hurt performance. See Table 1 for a summary of the effect of each of these changes. In each case we report the impact of these changes on Graph500 graphs of scales 15, 20 and 24. We chose these three scales because they were representative of three categories of L1 cache, L2 cache and global memory behavior.

At scale 24, most of the changes show smaller effects on the performance. This shows that the lack of locality for higher scale graphs becomes the dominating factor that impacts performance. This trend holds true for changes that did not improve performance as well; the changes are smaller at higher scales.

3.1.1 Warp-Level Thread Cooperation

To increase the performance of our implementation, we adapted an optimization mentioned in [6] that allows threads within the same warp to cooperatively process adjacencies of a node. With only 42 lines of additional code including comments, warp-level thread cooperation increased our

Variant	Scale 15		Scale 20		Scale 24	
Base	0.157		0.925		0.725	
Wide Bitmaps	0.155	-2%	0.885	-4%	0.710	-2%
Mult. V./Thr.	0.147	-6%	0.912	-1%	0.725	0%
S. D. Kernel	0.087	-44%	0.771	-17%	0.719	-1%

Table 2: The effect of changes that resulted in a performance decrease (GTEPS mean of 10 runs on a Graph500 graph and change relative to the best performing version from Table 1).

performance by about $16\times$.

We ran our benchmark on scales 15, 20 and 24 with only the warp-level thread cooperation optimization turned on, processing all adjacencies without launching child kernels, and we achieved 0.185, 0.686 and 0.564 GTEPS respectively, which is roughly a 15% decrease in performance compared to using both warp-level thread cooperation and child kernels (Table 1). This result shows that launching dynamic kernels for large adjacency processing does improve the performance of our algorithm.

3.1.2 Edge Sorting

When analyzing the profiles for our code we noticed poor memory behavior. This is to be expected given the sparse and irregular nature of graph algorithms, but we can hopefully make small improvements. Many vertices may point to the same vertex v , meaning the threads responsible for each of these vertices will each have to access v ’s entry in the parent map. We will get better performance if these accesses happen at the same time, and the likelihood of this occurring increases if we ensure threads process adjacent vertices in sequential order.

Sorting each vertex’s edges before doing the search improves the memory transactions per access ratio, which in turn leads to about a 8% overall improvement in performance at scale 20. Smaller graphs see a greater improvement, which is to be expected as they have better locality just by virtue of being smaller.

3.2 Failed Improvements

We discuss our experience with some changes that did not help here. Table 2 summarizes the performance impact of each of these changes.

3.2.1 Wide Bitmaps

Our implementation represents the frontier using a single bit per vertex that indicates whether the vertex is in the frontier. Updating this bitmap requires atomic operations as multiple threads can set bits in the same word at once. We had hoped that replacing the bitmap with an array of 32-bit values could gain some performance by removing the need for atomic operations.

This was not the case. We saw a performance decrease of 2%~4%, meaning the reduced overhead for atomic operations did not overcome the increased memory traffic caused by expanding the size of the bitmap.

3.2.2 Multi-Vertices per Child Kernel Thread

To increase throughput and decrease the number of blocks launched per child kernel, we experimented with increasing the *number of adjacent vertices processed per thread (NAPT)* in the child kernel. In our warp-level thread cooperation

processing stage, we were able to increase the performance of our algorithm by over 100% after we increased the number of adjacent vertices processed by each warp-level thread. We believed, therefore, that we might be able to achieve an even higher level of parallelism and further improve our performance by allowing a thread in the child kernel process more than one adjacency at a time.

We were able to reduce the total number of threads launched in the child kernel by setting $NAPT$ to greater than 1 but at the cost of performance decrease. The results in Table 2 show the performance impact of processing 2 vertices per thread.

Figure 1: BFS performance scaling for a selection of algorithms on a single GPU.

3.2.3 Single Driver Kernel

Our kernels discussed so far are able to perform one level of the BFS traversal without any input from the CPU. We tried to reduce the latency introduced by round trips between the CPU and GPU at each iteration by running the driver loop on the GPU as well. This led to a performance decrease of 44% on scale 15 and 1% on scale 24. It might be possible to combine the driver kernel and kernel1 together to improve performance, but we have not explored this.

4. SINGLE SOURCE SHORTEST PATHS

Since both BFS and SSSP algorithms can utilize a similar level-synchronized strategy for their traversals, it was easy to modify our BFS algorithm to perform SSSP calculations using dynamic parallelism (SSSP-DP). Instead of marking a neighbor as visited in each iteration, we compared the sum of the weight of the edge and the weight of the parent to the previously marked weight of the neighbor. If the new weight by visiting from the current parent is smaller than the recorded weight of the neighbor, the new weight is updated and the neighbor is marked to be visited in the next frontier.

5. EVALUATION

In addition to the results we have presented so far, we tested the performance of our algorithm on a number of graphs and compared them with existing implementations from the literature. Despite of our code’s simplicity, it performs competitively against other single-GPU BFS implementations.

Figure 2: BFS road network performance. Higher bars are better.

5.1 Experimental Setup

We performed our experiments on an Intel Xeon E5-2670 v3 12-core processor machine with 32 GB RAM and an NVIDIA Tesla K40 GPU, which includes 12 GB of RAM and 2880 CUDA cores.

We ran different BFS and SSSP implementations on both Graph 500 graphs and the USA road networks graphs including different algorithms included in the Lonestar GPU package as well as Ueno et. al. on Graph 500.

5.2 Breadth-first search Performance

Figure 1 shows how our implementation’s performance compares against Merrill et al.’s[6] and Ueno et al.’s[9] on Graph 500 graphs. Our implementation performs the best of the three through scale 19 at which point [6] overtakes ours.

As the graph gets larger, the spatial locality in memory decreases. We believe that after scale 19 the locality has decreased to the point where more TLB misses are occurring. The overhead introduced by launching more dynamic kernels could also have impacted performance, which explains our decrease in performance at larger scales. Adopting more techniques in [6] could alleviate this performance decrease.

Figure 2 shows our performance on a selection of road networks, where lonestar-wla, lonestar-wlc and lonestar-wlw represent different worklists used in the lonestar BFS algorithm collection. Note that none of the nodes in the road network have a high enough degree to take advantage of dynamic parallelism, which explains some of our relatively decreased performance.

5.3 Single source shortest paths Performance

Since atomic operations are required to compare and update the distances of each node in the graph for SSSP, SSSP algorithms are naturally more time consuming than traversing the same graph using BFS. We compared our timing against a collection of SSSP algorithms using different topologies and data structures in Lonestar GPU using graphs of the US road networks from the 9th DIMACS challenge [3] and the performance results can be seen in Fig. 3, where lonestar-wlc and lonestar-wln represent different worklists used in the lonestar SSSP algorithm collection.

The results suggest that our SSSP implementation lags behind some of the implementations in Lonestar that utilize advanced work queuing techniques. Due to the nature of the

Figure 3: SSSP road network performance

road network graphs, however, all the nodes are processed in the warp-level thread cooperation stage since none of the nodes in a road network graph has enough adjacent vertices to trigger child kernel processing. Therefore, our algorithm might not be able to achieve as much parallelism on road network graphs as on a Graph 500 graph.

6. RELATED WORK

We have already referred extensively in this paper to [6] as an exemplar of the best in class single-GPU BFS implementation. Our implementation adopts several techniques from this work. Wang and Yalamanchili [10] examine the efficiency of DP on a selection of problems such as the overheads of launching dynamic kernels. To decrease the cost of dynamic parallelism, Wang et al. [11] propose a more lightweight mechanism of dynamically adding blocks to the current kernel. We also compared our performance against [9], which focuses on using GPUs in a distributed setting on MPI. They use a vertex sorting technique, as in their earlier work [8], that is similar to our edge sorting optimization (Section 3.1.2). [5] developed a *warp-centric* programming model which allocated tasks at warp granularity to minimize divergence with warps. performance improvement. A recently published work by Davidson et. al. [2] proposes an SSSP algorithm that builds on the model and optimizations mentioned in [6] and reports up to 14x improvement on low-degree graphs and 20-60x improvement against a serial implementation of SSSP using Bellman-Ford.

7. CONCLUSION

We have shown that dynamic parallelism is a useful tool in simplifying the development of high performance GPU graph algorithms. Starting with a very simple algorithm, we can progressively apply additional improvements to yield better performance without increasing code complexity. At some scales our implementation even outperforms the one of the faster single-GPU BFS algorithms published to date (Lonestar GPU’s adaptation of [6]). The progressive manner in which we have built our algorithm leads to simpler, modular and highly maintainable code. In addition to simplifying code, CUDA’s dynamic parallelism serves as a tool to increase performance at a higher level of abstraction.

8. ACKNOWLEDGEMENTS

We thank the NVIDIA corporation for donating two K40 GPUs to experiment with. We thank Duane Merrill for

help with running b40c and Nate Husted for helping us troubleshoot synchronization issues.

This material is based upon work funded and supported by the National Science Foundation under Grant No. 1111888, and the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. References herein to any specific commercial product, process, or service by trade name, trade mark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by Carnegie Mellon University or its Software Engineering Institute. This material has been approved for public release and unlimited distribution. DM-0002795

References

- [1] A. Adinets. Adaptive Parallel Computation with CUDA Dynamic Parallelism, May 2014.
- [2] A. Davidson, S. Baxter, M. Garland, and J. Owens. Work-Efficient Parallel GPU Methods for Single-Source Shortest Paths. In *Proc. 28th Internat. Parallel and Distributed Processing Symposium*, pages 349–359, May 2014.
- [3] C. Demetrescu, A. Goldberg, and D. Johnson. 9th DIMACS Implementation Challenge: Shortest Paths, 2006.
- [4] P. Harish and P. J. Narayanan. Accelerating Large Graph Algorithms on the GPU Using CUDA. In *Proc. 14th International Conference on High Performance Computing*, pages 197–208. Springer, 2007.
- [5] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun. Accelerating CUDA Graph Algorithms at Maximum Warp. In *Proc. 16th ACM Symposium on Principles and Practice of Parallel Programming*, pages 267–276. ACM, 2011.
- [6] D. Merrill, M. Garland, and A. Grimshaw. Scalable GPU graph traversal. In *Proc. 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 117–128. ACM, 2012.
- [7] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang. Introducing the graph 500 benchmark. *Cray User’s Group (CUG)*, 2010.
- [8] K. Ueno and T. Suzumura. Highly Scalable Graph Search For the Graph500 Benchmark. In *Proc. 21st Internat. Symposium on High-Performance Parallel and Distributed Computing*, pages 149–160. ACM, 2012.
- [9] K. Ueno and T. Suzumura. Parallel distributed breadth first search on GPU. In *Proc. 20th Internat. Conference on High Performance Computing (HiPC)*, pages 314–323, Dec. 2013.
- [10] J. Wang and S. Yalamanchili. Characterization and analysis of dynamic parallelism in unstructured GPU applications. In *Proc. IEEE Internat. Symposium on Workload Characterization (IISWC)*, pages 51–60, Oct. 2014.
- [11] J. Wang, N. Rubin, A. Sidelnik, and S. Yalamanchili. Dynamic Thread Block Launch: A Lightweight Execution Mechanism to Support Irregular Applications on GPUs. In *Proc. 42nd Annual International Symposium on Computer Architecture*, pages 528–540. ACM, 2015.