Performance impact of dynamic parallelism on different clustering algorithms

Jeffrey DiMarco and Michela Taufer
Computer and Information Sciences, University of Delaware
E-mail: jdimarco@udel.edu, taufer@udel.edu

ABSTRACT

In this paper, we aim to quantify the performance gains of dynamic parallelism. The newest version of CUDA, CUDA 5, introduces dynamic parallelism, which allows GPU threads to create new threads, without CPU intervention, and adapt to its data. This effectively eliminates the superfluous back and forth communication between the GPU and CPU through nested kernel computations. The change in performance will be measured using two well-known clustering algorithms that exhibit data dependencies: the K-means clustering and the hierarchical clustering. K-means has a sequential data dependence wherein iterations occur in a linear fashion, while the hierarchical clustering has a tree-like dependence that produces split tasks. Analyzing the performance of these data-dependent algorithms gives us a better understanding of the benefits or potential drawbacks of CUDA 5's new dynamic parallelism feature.

Keywords: K-means, Divisive hierarchical clustering, CUDA 5.0.

1 INTRODUCTION

CUDA's dynamic parallelism is a feature that aims to improve performance of data-dependent functions on Graphics Processing Units (GPUs) and is supported in the new CUDA 5 release. GPUs have been extensively used for accelerating scientific applications, but a significant overhead for communication during runtime is required when the associated algorithms exhibit data-dependencies. The communication arises because only the CPU can launch work on the GPU, so the GPU needs to ask permission from the CPU to start new algorithm iterations. The CPU must determine whether and how to launch subsequent computations based on the particular algorithm dependencies. Dynamic parallelism aims to extend the CUDA programming model so that the work can be updated and continued without CPU intervention. The control flow between CPU and GPU for CUDA applications can be greatly reduced resulting in increased performance and ease of use for the programmer [1].

Relevant classes of algorithms that can potentially benefit from dynamic parallelism when executed on GPUs are the clustering algorithms. Clustering algorithms are extensively used in data mining, for example, and can be applied to many applications in scientific computing. Clustering refers to taking a large set of objects or elements and organizing them into collections based on their similarity. This similarity can be measured with many metrics; commonly, it is simply the distance between elements in space. In our work, we focus on two types of clustering: K-means and divisive hierarchical clustering. K-means exhibits a linear dependence, while divisive hierarchical clustering exhibits a binary tree-based dependence [2]. These two methods have different types of data dependence and thus can give us a broader perspective of how dynamic parallelism can affect performance.

In this paper, we investigate how CUDA's dynamic parallelism affects the performance of K-means clustering and hierarchical divisive clustering. We provide evidence on how different types of data dependence are affected in terms of performance. In Section 2, we describe how these two clustering algorithms work mathematically. In Section 3, we go into detail about our CUDA implementations that apply CUDA 5.0's dynamic parallelism. In Section 4, we outline the testing environment and data followed by our results. Finally, in Section 5 we present our conclusions and what they imply for future research.

2 BACKGROUND

2.1 K-means clustering

K-means is a clustering algorithm used in several applications of data mining, machine learning, and scientific applications [3]. The K-means algorithm takes a set of data points as input and a number k of clusters to partition the data points [4]. The *k* clusters are centered at the *k* centroids, which are the points used for assigning data elements to each cluster. Initially, each centroid is assigned randomly; at each iteration the centroids move to the average of all the data points that belong to the associated cluster. A similarity function is used in order to determine to which centroid each data point should be allocated. Generally, this similarity function is simply the Euclidean distance, but other possible similarity functions can be defined and used. The general process is to measure the similarity of each of the data points to the *k* centroids and assign each point to the closest one. After assignment, the centroids' position is changed to the average of all data points belonging to it. This process repeats until a convergence condition is met. Most commonly the convergence occurs when the centroids have stabilized and no longer change positions. Due to limitations of the algorithm, however, convergence might not be achieved. In this case, the stopping criteria is having the algorithm run through as many iterations as necessary to obtain a solution within an acceptable error boundary. Algorithm 1 shows the workflow for K-means.

Algorithm 1: General overview of the K-means algorithm.

```
1     Given data set X, set of k centroids C
2     Randomly select k centroids from X to populate C
3     Until convergence
4       For all data x in X
5         Find closest centroid to x
6         Associate point x to centroid c
7       End for
8       For all centroids c in C
9         Compute new average of members of c
10      End for
11    End until
```

2.2 Hierarchical clustering

Hierarchical clustering takes groups of elements and organizes them into a hierarchy [5], which can be created either with agglomeration by combining clusters, or divisively by recursively splitting the entire dataset. Agglomerative clustering considers each point in a data set as a cluster containing one element. The two closest clusters are combined and then viewed as one containing the sum of their elements until all clusters have been combined into a subset of clusters whose elements are similar. Divisive clustering, on the other hand, begins with a single cluster containing all of the elements in a data set and splits it into two. Afterwards, the clusters that remain are split into two recursively until a subset of clusters, each with similar elements, is identified. The data under consideration changes with each iteration as the hierarchy grows. In agglomerative hierarchical clustering the number of clusters to be considered is reduced after each iteration, while divisive hierarchical clustering has an increase in the number of clusters after each iteration. Our work focuses on divisive clustering because of the data dependency it brings. Each task generates two subsequent tasks,

Algorithm 2: General algorithm for divisive hierarchical clustering.

```
1     Given data set X
2     Until convergence
3       // Split X in two clusters (k=2)
4       For each data x in X
5         Assign x either to cluster X1 or to cluster X2
6         Store memberships into global structure
7       End for
8       Repeat on data in X1 from step 1
9       Repeat on data in X2 from step 1
10    End until
```

which increases the amount of communication to the CPU as the algorithm runs to completion. The computation has a tree-like structure similar to a divide-and-conquer algorithm; thus dynamic parallelism is expected to provide some performance gains. The workflow is shown in Algorithm 2

3 METHODOLOGY

3.1 K-means without dynamic parallelism

To create a baseline for GPU performance we developed an implementation of K-means that takes advantage of the computational parallelism of GPUs but not dynamic parallelism. K-means performs two main tasks that are sent to the GPU for acceleration (i.e., data assignment and centroid updating, as described in Section 2). At each iteration of the main loop, the CPU issues both of these tasks as kernels to the GPU until the convergence condition is met. The basic workflow is presented in Algorithm 3.

Algorithm 3: K-means algorithm on the GPU.

```
On CPU                                      On GPU

Given data set X, set of k clusters C
Initialize C to points in X
Until convergence
   Launch data assignment kernel    ------> GPU thread: For each data x in X
                                                compute closest centroid to x
                                    <------ End for
   Launch centroid update kernel    ------> GPU thread: For each data x in X
                                                accumulate x to its cluster c
                                                increment total members for its cluster c
                                    <------ End for
   For all cluster c in C
      Divide accumulation by total members
   End for
End until
```

The first task is to determine the closest centroid to each data point. The first GPU kernel takes the vector of data and a membership vector as input. The membership vector holds the closest centroid for the corresponding element in the data vector. The GPU kernel launches a thread for each data point to calculate the minimum of the distances to a centroid. The centroid that each thread computes is recorded into the membership structure.

The second GPU task takes the centroid vector, data vector, and the membership vector as inputs and reassigns centroids using a reduction schema to compute partial results. The GPU kernel uses a thread for each element in the membership vector and a thread block as a running sum for each centroid. Whenever a data point belongs to a centroid, its coordinates are accumulated into a running sum, which is then accumulated between thread blocks and divided by the total number of data points belonging to each centroid. Atomic add functions ensure that accumulation is done correctly since some data points in a thread block belong to the same centroid. The total sum of data points belonging to a centroid is divided by the total number of data points which results in a new average for each centroid. The convergence condition is met when the centroids have stopped moving, which is the case if the membership array did not change during the data assignment kernel. After the two kernels run, the GPU transfers a Boolean flag back to the CPU to determine when to stop issuing work on the GPU.

3.2 K-means with dynamic parallelism

Implementing K-means with dynamic parallelism does not change the code for the kernels. The control is shifted to the GPU instead, which removes the memory transfer of the Boolean flag to the CPU, as described above in Section 3.1. The main loop uses the kernels for assigning the data to centroids and updating centroid positions, and the Boolean flag is checked for convergence by the GPU instead of the CPU after the kernels run. In this scenario, the GPU issues work instead of the CPU. As a result, transferring of data and synchronizing with the CPU no longer occurs; instead, the GPU synchronizes without CPU interaction. The CPU is used only for initialization, then the work stays on the GPU completely. Another important aspect of applying dynamic parallelism is the improvement in readability and programmability for the developer. Dynamic parallelism removes the burden of memory management after initialization and helps GPU programming to look even more familiar to CPU programming [1].

3.3 Hierarchical clustering without dynamic parallelism

We developed a CUDA implementation of divisive hierarchical clustering without dynamic parallelism to use as a baseline for performance when applying dynamic parallelism. The hierarchy is stored in a matrix that we refer to as the membership matrix. Each row of this matrix holds the centroid that each data element belongs to so that once a convergence is reached the entire hierarchy is stored. The data can be reorganized in memory during runtime, but memory coalescence needs to be preserved to allow multiple memory-reads for a single request. In order to accomplish this, we developed a partitioning kernel to maintain contiguous memory accesses. Without partitioning, contiguous memory accesses would not be guaranteed. The general idea is to first split the input data in half and then to partition each half into contiguous chunks. The final step is to repeat the process on the two newly created groups. Algorithm 4 helps visualize this workflow.

The points are first split using K-means where *k* is equal to 2 with the GPU implementation described above in Section 3.1. K-means finds the most suitable pair of clusters to describe the input data. After separating the data set logically into two clusters, the data needs to be organized in memory. Subsequent iterations of the algorithm operate on each of the two clusters created after the split, so it is best that each piece is contiguous. The partitioning kernel is done on the GPU to maximize performance of the algorithm.

To partition a large chunk of data in parallel, a large number of threads cooperate to rearrange piecewise from the original data array. We use two arrays such that each thread can read asynchronously from one and write asynchronously to the other. The partitioning kernel delegates a data point to each thread and operates on multiple groups of 32 to correspond with the warp size. The input to the partitioning kernel is the section of the data vector and the corresponding membership array which holds the assigned centroids. The membership array is then used to determine where to place the data elements so that the output is a contiguous chunk of those elements.

One of the issues that arises when partitioning in parallel is determining where each data point must be written to such that all threads cooperate in their asynchronous fashion. This is accomplished with a combination of warp voting and atomic add operations. An atomic add function is used to determine where each thread block begins writing by referring to the return value, which is the starting point for the current thread block to write. The current thread block then adds its number of writes, as determined by the warp voting functions, to ensure that the next starting point is correct for each subsequent group of threads. The warp functions are __ballot() and __popc() to count the amount of points belonging to each of the two clusters. Each thread block then knows how many points it must write to each of the respective clusters. Once the data has been assigned to a cluster and correctly partitioned for the next split, the membership matrix is written with the corresponding elements' memberships.

Algorithm 4: Divisive hierarchical clustering on the GPU

```
        On CPU                                    On GPU
1       Given data set X
2       Until convergence
3          // Split X by using K-means
4          Launch partitioning kernel    ───────►  GPU thread for each data x in X
5                                                     Partition into corresponding
6                                                     membership of X1 or X2 in M
7                                        ◄───────  End for
8          Repeat on data in X1 from step 1
9          Repeat on data in X1 from step 1
10      End until
11      Return membership matrix M
```

3.4 Hierarchical clustering with dynamic parallelism

As in the case with K-means, implementing dynamic parallelism for divisive hierarchical clustering does not change the code for the kernels used. The kernels still perform the same operations, but the communication to the CPU is removed entirely. With dynamic parallelism, two subsequent tasks can execute concurrently on different chunks of the same data vector. Without dynamic parallelism, the CPU serially checks and issues tasks to the GPU. Calling kernels from the GPU is a more natural workflow and works well with divide-and-conquer algorithms such as the divisive hierarchical clustering. In terms of performance, the advantage to adding dynamic parallelism is that the GPU can determine if the children processes should be further divided; it this is the case, the GPU places them in separate work streams without the need for CPU intervention. Both the added concurrency provided by the work streams and the removed CPU communication gives the GPU more parallelism and higher utilization.

4 TESTING

4.1 Overall goal

The goal of our experiments is to determine whether different clustering algorithms have performance gains when used with dynamic parallelism. At first it would seem that GPUs can emulate the host and launch work locally, which would remove the majority of memory transfers from host to device; however, the developer needs to understand what dynamic parallelism can offer for said performance gains.

4.2 Platform and data setup

All of our tests were run on a machine with a Quad-core Intel Xeon E5520 @ 2.27 GHz, 24GB memory, and the NVIDIA K20c GPU @ 706 MHz with 5GB GDDR5 memory @ 2.6 GHz. The data used were arrays of 3-dimensional single precision floating point numbers generated using a normal distribution.

In our experiments for K-means clustering, we considered different scenarios for our input data. We looked at different values for the number of clusters present, the number of clusters expected, and the total number of data points. The data was distributed normally among clusters with a weighting scheme where the clusters can be either of equal size or weighted. We wanted to determine if the data distribution affect performance as well as to isolate the associated effects of dynamic parallelism. The size of the data that our tests considered is 16k, 32k, 64k, 128k, and 256k points. For the number of expected clusters *k* we considered values of 1, 2, 4, 8, 16, 32, 64, and 128. Finally, the number of "actual" clusters that the data was split in was 1, 2, 4, 8, 16, 32, 64, and 128. For each of the actual clusters we also wanted to test the effects of their size. To do this we used evenly distributed clusters along with a weighting scheme where one of the clusters contained half of the data. The data was generated using a normal distribution, and each permutation of input size, actual clusters, and *k* expected clusters was tested and averaged over several runs.

For consistency we used the same data sets with divisive hierarchical clustering. The weighted clusters were of more interest in this case because the size of the recursions is expected to be more heavily biased towards the larger clusters. We used the same input sizes of 16k, 32k, 64k, 128k, and 256k points. We distributed the points among 1, 2, 4, 8, and 16 clusters that were both weighted and evenly split. The data was generated using a normal distribution for each of the clusters. The goal of these tests was to see how the size, distribution, and weight of the data points affected performance. Figure 1 shows a few examples of how the data was distributed.

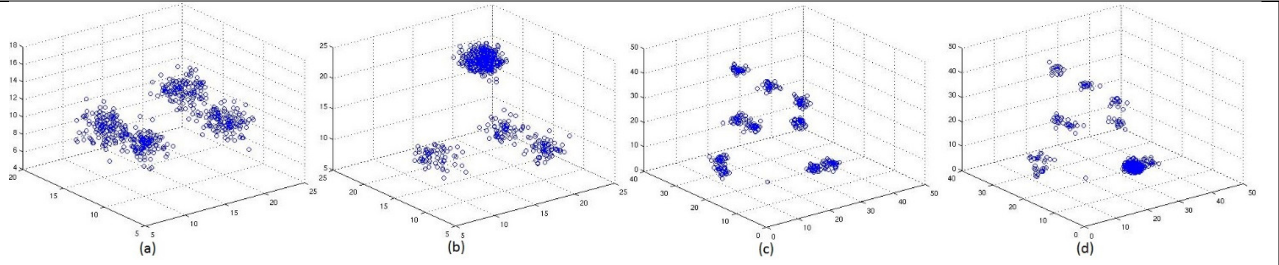

Figure 1: Different input data used for our testing, i.e., 4 clusters evenly distributed (a), 4 clusters with 1 cluster heavily weighted (b), 16 clusters evenly distributed (c), and 16 clusters with 1 cluster heavily weighted (d).

4.3 K-means

The data dependency for K-means between iterations is sequential, which means the algorithm issues work as long as the previous work did not meet the convergence condition. This also means that the work cannot execute concurrently, and an iteration must run to completion before the following iteration can begin. The CPU must call cudaDeviceSynchonize() as usual, but dynamic parallelism also requires that kernel calls from the device call cudaDeviceSynchronize() before issuing child kernels. Therefore, there is a barrier regardless of whether we are launching work from the CPU or the GPU. The only difference is that there is a memory transfer of a Boolean flag when launching the kernels from the CPU, which is used to determine convergence. At first we expected the elimination of a small memory transfer, which takes ~10-100 microseconds, to produce a performance gain. Contrary to our expectation, our tests showed that there is a slight performance loss regardless of the number of *k* centroids, actual clusters, or data size. We tested exhaustively for all of our input data and saw similar outcomes. Unfortunately, we observe that the overhead for synchronizing and launching on the device is more costly compared to the CPU execution. Figure 2.a presents our results by averaging the times for each of the several scenarios. We observe a combined performance loss of 7.7% averaged across all cases.

4.4 Divisive hierarchical clustering

Our experiments for divisive hierarchical clustering were timed and averaged over five runs; the observed results are shown in Tables 1 and 2. Each run used a uniquely generated data set that was used for the implementations both with and without dynamic parallelism. Using the same data sets between implementations gives us relative speedups but causes a slight variability in the trends as parameters change. We observe a constant factor speedup as the input size increases for divisive hierarchical clustering as seen in Figure 2.b. Similar to our K-means results, we did not discover any effect on performance when different data distributions are used. Tables 1 and 2 show the time (in seconds) taken for different data sizes and clustering scenarios. For example, 128K data points across all clustering schemes has an average increase in performance factor of 2.19 times speedup.

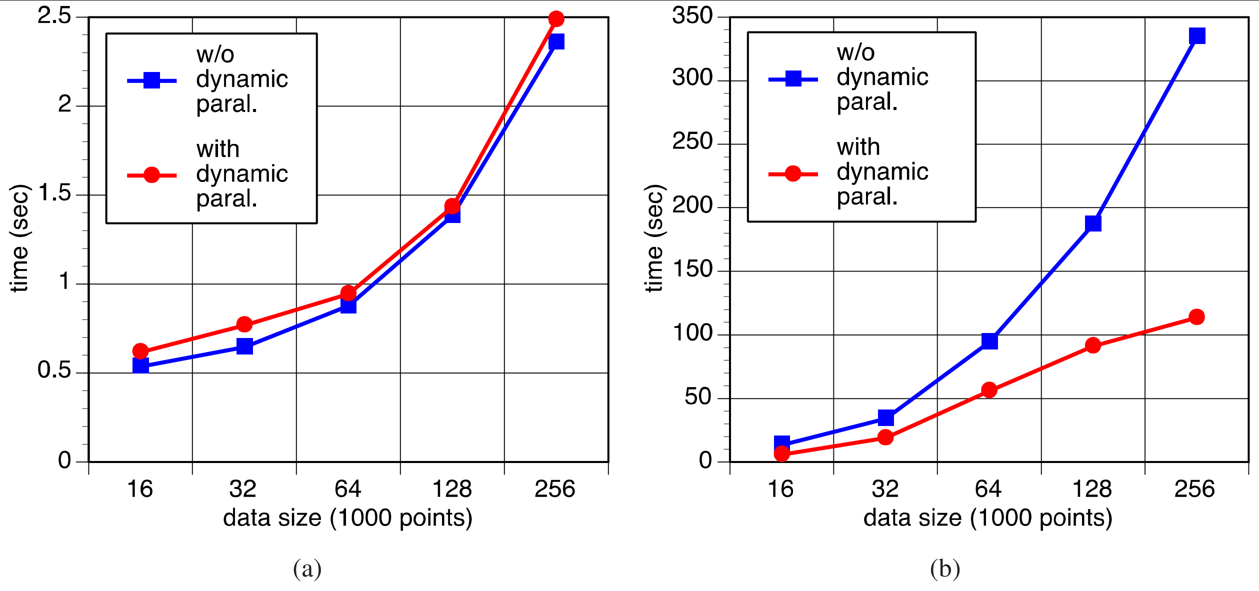

Figure 2: Average run times for each input size for K-means with and without dynamic parallelism (a) and for divisive hierarchical clustering with and without dynamic parallelism (b).

The speedups we observed varied a notable amount, which we hypothesize to be caused by the input data; nonetheless, the speedups occurred for every experiment we ran. The tables indicate that there is a noticeable variance between runs and cases, yet it is very clear that there is a significant increase in performance regardless of the case. We could not find any cases in which dynamic parallelism had a performance loss for divisive hierarchical clustering.

4.5 Discussion

One of the biggest challenges that dynamic parallelism aimed to solve is the loss of concurrency due to the CPU-GPU synchronizations, for example, in algorithms with some level of data dependencies. Since hierarchical clustering is a divide-and-conquer method, we were able to utilize dynamic parallelism for concurrent execution of computation. The results of our experiments show that implementing dynamic parallelism for this clustering algorithm increases performance by 1.78 times up to 3.03 times. We were also able to observe that the characteristics of the input data (i.e., how the points are distributed) do not affect the total performance gains in our tests.

Table 1: Average times for divisive hierarchical clustering without dynamic parallelism. Note that "E" refers to even clusters where the data points are spread evenly between all of them and "W" refers to weighted clusters where one of the clusters contains half of the points and the remaining data points are split amongst the rest.

Data Pts	2E	2 W	4E	4 W	8E	8 W	16E	16 W
16,384	13.80	13.41	13.49	13.78	13.10	13.30	13.81	13.46
32,768	36.48	35.51	33.17	31.84	34.26	31.15	34.89	34.46
65,536	78.75	87.17	97.58	96.51	107.72	102.14	111.93	93.16
131,072	152.82	167.40	174.55	199.14	210.00	225.65	210.32	200.02
262,144	285.71	289.57	364.18	309.35	366.73	417.20	409.73	337.15

Table 2: Average times for divisive hierarchical clustering with dynamic parallelism. Note that "E" refers to even clusters where the data points are spread evenly between all of them and "W" refers to weighted clusters where one of the clusters contains half of the points and the remaining data points are split amongst the rest.

Data Pts	2E	2 W	4E	4 W	8E	8 W	16E	16 W
16384	5.85	5.86	5.88	5.81	5.82	5.90	5.83	5.84
32768	20.94	19.45	20.86	15.55	19.01	18.05	19.95	17.36
65536	42.23	48.87	61.43	59.29	65.63	64.09	69.60	57.12
131072	55.65	67.56	84.23	121.67	106.81	121.67	99.96	117.14
262144	71.08	81.61	118.30	130.34	128.44	154.80	123.76	144.60

5 FUTURE WORK

One of the largest limitations of dynamic parallelism is the depth of child kernels in applications that evolve into tree-like structure of tasks. The depth refers to the maximum number of subsequent child kernels that can be called. There is a memory overhead on the GPU that limits the amount of child kernels to 24. If the programmer does not know how deep the iterations of a particular algorithm can go, then she needs to compromise accordingly. One solution to this problem is to revert to an implementation without dynamic parallelism once the depth limit has been reached. For our work, we focused solely on the performance of adding dynamic parallelism, but there are certainly applications that need to be mindful of the depth barrier. This could be a lucrative area for further research because dynamic parallelism is still in its infancy, and future research that explores solutions to the limits of dynamic parallelism is in need.

6 CONCLUSION

Dynamic parallelism lends itself to divide and conquer methods and facilitates GPU programming. Nested kernels are now a reality that can be taken advantage of as accelerators become more prevalent. In this paper we showed how dynamic parallelism can positively impact performance of the divisive hierarchical clustering algorithm. We observed speedups between 1.78 times and up to 3.03 times for data sets between 16K points and 256K points. The added concurrency is provided because of the tree-like data dependency that the algorithm brings. For K-means, however, we saw a slowdown in the performance. We observed a 7.7% slower runtime when applying dynamic parallelism across data sets of 16K up to 256K. We can conclude that although dynamic parallelism is not the solution for all GPU programming, it is nonetheless an important feature that helps push the boundaries of computational parallelism.

ACKNOWLEDGMENTS

Research supported by Air Force Small Business Technology Transfer (STTR) funding.

REFERENCES

[1] NVIDIA, Cuda Dynamic Parallelism Programming Guide. URL: http://docs.nvidia.com/cuda/pdf/CUDA_Dynamic_Parallelism_Programming_Guide.pdf (August 2012).

[2] Steinbach, M., Karypis, G., and Kumar, V., "A comparison of document clustering techniques," KDD Workshop on Text Mining, 400(1), 525-526 (2000).

[3] Jain, A. K., "Data clustering: 50 years beyond K-means.," *Pattern Recognition Letters,* 31(8), 651-666 (2010).

[4] Farivar, R., Rebolledo, D., Chan, E., and Campbell, R., "A parallel implementation of k-means clustering on GPUs," In *Proceedings of International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA),* 340-345 (July 2008).

[5] Chang, D., Kantardzic, M., and Ouyang, M., "Hierarchical clustering with CUDA/GPU," In *Proceedings of the 22nd International Conference on Parallel and Distributed Computing and Communication Systems (ISCA*), 7-12 (2009).