Characterization and Analysis of Dynamic Parallelism in Unstructured GPU Applications

Jin Wang
Georgia Institute of Technology
Atlanta, Georgia, USA
Email: jin.wang@gatech.edu

Sudhakar Yalamanchili
Georgia Institute of Technology
Atlanta, Georgia, USA
Email: sudha@ece.gatech.edu

Abstract—**GPUs have been proven very effective for structured applications. However, emerging data intensive applications are increasingly unstructured – irregular in their memory and control flow behavior over massive data sets. While the irregularity in these applications can result in poor workload balance among fine-grained threads or coarse-grained blocks, one can still observe dynamically formed pockets of structured data parallelism that can locally effectively exploit the GPU compute and memory bandwidth.**

In this study, we seek to characterize such dynamically formed parallelism and and evaluate implementations designed to exploit them using CUDA Dynamic Parallelism (CDP) - an execution model where parallel workload are launched dynamically from within kernels when pockets of structured parallelism are detected. We characterize and evaluate such implementations by analyzing the impact on control and memory behavior measurements on commodity hardware. In particular, the study targets a comprehensive understanding of the overhead of current CDP support in GPUs in terms of kernel launch, memory footprint and algorithm overhead. Experiments show that while the CDP implementation can generate potentially 1.13x-2.73x speedup over non-CDP implementations, the non-trivial overhead causes the overall performance an average of 1.21x slowdown.

I. Introduction

General purpose graphics processing units (GPUs) have demonstrated significant performance improvements for structured, data intensive scientific applications such as molecular dynamics [1], physical simulations in science [2], options pricing in finance [3], and ray tracing in graphics [4] to name a few. This has been due in no small part to the multidimensional grid structured organization of parallel thread blocks in the bulk synchronous parallel (BSP) model supported by NVIDIA's CUDA and industry standard OpenCL programming languages. Threads are mapped naturally into 1D-3D array data structures found in such applications and enable harnessing the substantial memory and compute bandwidths of GPUs.

However, the explosive growth of "big data" is being driven in large part by the accumulation of relational, semi-structured, and heterogeneous data types in all sectors of the economy. The emerging data intensive applications operating over these data sets are increasingly irregular in their memory and control flow behaviors while the massive parallelism is hierarchical, time-varying, and workload dependent. This raises important questions of how data intensive applications that operate on irregular data structures like trees, graphs, relational data, and adaptive meshes can be effectively parallelized and mapped to GPUs to harness their compute and memory bandwidths. This paper reports on the results of a study seeking to answer some of these questions.

We first observe that during execution these applications exhibit pockets of *dynamically* formed structured parallelism. In principle, when these pockets occur they can locally exploit GPU compute and memory bandwidth effectively. The CUDA Dynamic Parallelism (CDP) model offers mechanisms for nested kernel launch to exploit such pockets of parallelism. However, improving the expressiveness and simplification of the programming models comes at some expense in performance. Our goal has been to understand, characterize and report, on these tradeoffs. Specifically, this paper seeks to make the following contributions.

1) We show that pockets of structured data parallelism are dynamically formed in these emergent unstructured applications.
2) Eight benchmark applications are implemented using CUDA Dynamic Parallelism wherein nested (parallel) kernels can be launched dynamically when pockets of parallelism are detected.
3) We analyze the control flow and memory behavior of the CDP implementations and compare them to equivalent, non-CDP (i.e., traditional BSP) implementations.
4) We seek to characterize the occurrence of dynamic parallelism, the ability of CDP to effectively exploit it in terms of algorithm, runtime and memory footprint overhead.

To the best of our knowledge, this is the first paper to comprehensively analyze CDP and quantify the scope of application and overheads. The rest of the paper is organized as follows. Section 2 introduces the background of NVIDIA Kepler GK110 architecture and CUDA programming model used in this paper. we also describe briefly the CUDA Dynamic Parallelism technique supported on GK110. Section 3 provides the concept of unstructured applications and dynamically formed pockets of parallelism. Section 4 proposes the implementation of unstructured applications with CDP. Section 5 describes the methodology used in the study by showing experiments platforms, metrics and benchmark applications for measurement and analysis. Section 6 characterizes and analyzes the unstructured applications with several metrics and evaluates the CDP overhead. Section 7 reviews the related work, followed by the conclusions in Section 8.

II. Background

This section briefly introduces the NVIDIA GPU architecture and the CUDA Dynamic Parallelism (CDP) programming model. We focus on the Kepler GK110 architecture since it is the only architecture available that supports CDP. However, the methodology and analysis also apply to new architectures and programming models that support device kernel launch [5].

A. GPUs and CUDA Programming Model

The NVIDIA Kepler GK110 architecture [6] comprises of several Streaming Multiprocessors (SMX) units, each of which features 192 single-precision CUDA cores, 64 double-precision units, 32 special function units and 32 load/store units. It also includes 64K, 32-bit registers and 64KB scratch-pad memory that can be used either as a L1 cache or shared memory. CUDA [7] is the programming model introduced by NVIDIA for its GPUs. In CUDA, a program is expressed as a set of parallel kernels in which threads are grouped together into thread blocks or Cooperative Thread Arrays (CTAs) and then into 1D-3D grids. A CTA is divided into 32-thread warps as the basic execution unit on GPUs. Each SMX in the GK110 has four warp schedulers and eight instruction dispatch units, allowing four warps to be issued and executed concurrently. The warp scheduler chooses from a pool of ready warps according to some scheduling policy and executes them on the SMX. Multiple warps are interleaved to hide memory latency. Control flow divergence [8] occurs when threads within a warp take different execution paths, resulting in serialization of thread execution, and low SIMD lane utilization. Non-coalesced memory accesses happen when the memory data accessed by threads within a warp do not fall into a contiguous 128 byte block, and therefore results in multiple memory transactions. Memory divergence [9] occurs when memory references experience different latencies due to bank conflicts or cache misses, or non-coalesced accesses. Memory divergence increases the memory access latency for the entire warp.

B. CUDA Dynamic Parallelism

Cuda Dynamic Parallelism (CDP) [10] is the new functionality provided by the Kepler GK110 architecture. It allows kernels to be launched from GPU device without going back to the host. The kernel, block or thread that initiates the device launch is the parent and the kernel, block or thread that is launched by the parent is the child. CDP allows explicit synchronization between the parent and the child through a device runtime API `cudaDeviceSynchronize`. Launches can be nested from parent to child, then child to grandchild and so on. The deepest nesting level that requires explicit synchronization is referred as the syncrhonization depth. The maximum syncrhonization depth supported on GK110 is 24. Parent will be suspended and yield to child kernels if explicit synchronization is required. If no explicit synchronization is specified, there is no guarantee of the execution order between the child and parent. Concurrent execution of child kernels are possible but not guaranteed, depending on available GPU resources. GK110 architecture supports 32 concurrent kernel execution including both the host-launched and device-launched kernels. Parent and child kernels have coherent access to global memory with full consistency only at the point when parent kernels launch child kernels or the explicit synchronization is requested. Shared memory and local memory are exclusive for parent and child kernels and are invisible to each other.

When a child kernel is launched, the parameter buffer pointer of the kernel is retrieved through the device runtime API `cudaGetParameterBuffer`. Then the argument values are stored in the parameter buffer and the kernel is launched by calling `cudaLaunchDevice`. After that, the device runtime manager appends the child kernels to an execution queue and dispatches the kernel to SMXs according to a certain scheduling policy. The CDP *kernel launching overhead* comprises of kernel parameter parsing, calling `cudaGetParameterBuffer` and `cudaLaunchDevice`, as well as the process that device runtime manager setups, enqueues, manages and dispatches the child kernels.

III. Dynamic Parallelism in Unstructured Applications

In this section, we provide a brief description of the behavior of unstructured applications as viewed for the purpose of this paper and in particular the notion of dynamically formed pockets of structured parallelism.

A. Impact of Unstructured Applications

Unstructured applications are applications that possess irregular control flow and memory behavior over massive unstructured data sets, e.g., tree, graph and adaptive meshes. Control divergence, memory access patterns, and degree of parallelism are dynamic, time varying, workload-dependent, and difficult to predict. The CUDA and OpenCL programming model is structured around massively parallel threads grouped into CTAs organized into 1D to 3D grids. Data parallel computations over multidimensional arrays of data fit well within this model where each thread can be mapped into a logically contiguous partition of the data set.

However, when data is not structured in this fashion, data-dependent computations result in poor workload balance among fine-grained threads or coarse-grained CTAs which leads to increased control flow divergence, poor memory system performance and low SMX utilizations. For example, vertex expansion is a common operation used by many graph algorithms. When a thread is assigned a vertex, vertex expansion can require each thread to deal with a different number of edges, sometimes 1-2 orders of magnitude different. Therefore, the computations in threads within a CTA, or across CTAs are very imbalanced leading to low utilizations. Further, vertex expansion can generate (depending on the choice of data structure) non-coalesced memory accesses due to the lack of spatial locality across adjacent vertices leading to multiple memory transactions and increasing memory divergence, e.g., threads finish memory instructions at different times.

Finally, another common strategy for handling unstructured data is to use loop iterations within each BSP thread to access non-contiguous data elements. This too leads to increased memory divergence and load imbalance. As data structures become more diverse, the mapping of data to threads becomes more complex and variance in memory access patterns and control flow grow accordingly.

B. Dynamically Formed Pockets of Structured Parallelism

Structured memory accesses and uniform control flow make the best use of the computational and memory bandwidth of GPUs. In spite of the observations in Section III-A, one can observe **D**ynamically **F**ormed pockets of structured data **P**arallelism (DFP) in these applications that can locally effectively exploit the GPU compute and memory bandwidth. For example, in vertex expansion common data structures used in graph problems (e.g. Compressed Sparse Row or CSR [11]) store neighbors of one vertex in consecutive addresses and the memory access can be coalesced when neighbors are explored in parallel. In adaptive mesh refinement used in combustion simulations, certain parts of the mesh will be refined in parallel into a finer grained mesh creating hierarchical nested grid structures each of which may be of different dimensions. In general, we observe that DFP commonly occurs in one of the following two patterns:

Static Data Structure Traversal. Applications have irregular but statically defined data structures while the algorithms that traverse them encounter varying degrees of parallelism. Graph and tree traversal algorithms such as breadth first search (BFS) are examples in this category.

Dynamic Data Generation. The application data structures themselves are generated during execution and their form and extent are themselves data dependent. For example, combustion simulation (adaptive mesh refinement), tree generation (indexing) and the relational JOIN operator all start from an initial data set and dynamically generate new irregularly structured data sets in parallel.

Given the preceding view of the behaviors of unstructured applications the remainder of paper addresses the characterization and analysis.

IV. Implementation using CUDA Dynamic Parallelism

We propose to use CDP to implement the unstructured applications and handle the DFP throughout the program. The implementation launches a new kernel from the device by the parent thread when DFP is detected. A common code structure is shown in Listing 1, where a parent thread checks some conditions and determine whether new parallel workload should be launched through `childKernel` either to traverse a new portion of the data structure or to generate new data sets. A straightforward understanding of CDP implementation is that it replaces the parallel loops performed by threads in the non-CDP implementation by child kernels. We specifically consider the following aspects in the implementations.

```
//executed by each thread in parent kernel
threadData = getData(threadId);
if(condition(threadData))
    childKernel<<<CTAS, THREADS>>>
        (threadData, ...);
```

Listing 1. Common code structure that handles DFP with CDP

Parallel computation workload. As defined in DFP, the newly launched kernels handle the parallel computation workload that is discovered dynamically by a parent thread at runtime. This is in comparison to the unbalanced GPU implementations where each thread could use a loop with different iteration count to deal with the computation. Since parent threads only have to issue a child kernel and child kernels handle only parallel computation with little or no control divergence, the CDP implementation can achieve higher GPU utilization. However, it should be noted that sometimes there is not enough parallelism to launch a child kernel. For example, if the neighbor degree in vertex expansion problem is less than the warp size, SIMD lanes cannot be fully utilized if a new kernel is launched for expanding that vertex. In this case, the computation will still be left to the parent kernel.

Memory access patterns. The memory access pattern in a CDP implementation can be different from a non-CDP implementation. The memory addresses that are accessed by one thread in different loop iterations in a non-CDP implementation are now accessed by contiguous threads in a child kernel using one memory instruction. This could effectively change the number of coalesced memory accesses as well as cache hit rate.

Recursion. Recursive kernel launch is possible with the CDP support on GPUs. For some of the unstructured applications, it is necessary to recursively launch new computation dynamically. Non-CDP implementation tends to either convert the recursive algorithm to loop iterations or manage stack-based data structure at both the host and the device. The CDP implementation simply calls the same kernel recursively.

Concurrent kernel execution. Child kernels are launched independently from each other and can be executed concurrently. The implementation use one stream for each child kernel launch. Although no concurrency can be guaranteed from the perspective of GPU architectural support of CDP [10], the use of CUDA streams can increase the possibility to the most extent.

Child kernel configuration. A common practice for GPU programming is to partition workload between CTAs and then between threads. The same argument holds true for child kernels in our implementations. We experiment using different CTA sizes and grid sizes to generate optimal performance. CTA sizes should be multiple of 32 to eliminate any intra-warp thread divergency for child kernels. When the block size is not a multiple of 32, the remaining threads are executed by the parent kernel. This is analogous to the loop transformation that unrolls a loop k times by creating two loops - one that is unrolled k times and one that has loop bounds of $(N \ mod \ k)$ where N is the loop bound.

Shared memory. Current form of CDP does not allow the child kernel to access the shared memory declared by the parent kernel. Therefore, if the dynamically launched child kernel needs to access the data stored in the shared memory, CDP implementation either pass the data value directly as the kernel argument or dump them into global memory. The former solution can only deal with small number of arguments and the latter solution could introduce huge memory and runtime overhead.

Synchronization. CDP supports explicit synchronization between parent and child kernels at the substantial cost of both execution time and memory footprint. Therefore, the implementation would avoid using synchronization as much as possible. However, there are still a few cases that synchronization is necessary to conserve either temporal or spatial ordering consistency.

V. Methodology

A. Experiment environment and Metrics

We perform experiments on multiple GPUs with Kepler GK110 architectures, including NVIDIA Tesla K20c, Geforce Titan and Tesla K40. Table I shows the features of these GPUs.

TABLE I. NVIDIA GPUs USED FOR EXPERIMENTS

	Tesla K20c	Geforce Titan	Tesla K40
SMX	13	14	15
Cores	2496	2688	2880
Clock Frequency (MHz)	706	837	745
Global Memory Capacity (GB)	5	6	12
Memory Bandwidth (GB/s)	208	288	288

We experiment with both non-CDP implementations and CDP implementations of the unstructured applications. The CUDA 5.5 toolkit is used including the nvcc compiler and the runtime library. For CDP implementations, the compiler also links against cuda device runtime library, i.e. `-lcudadevrt`. We use the CUDA Profiler NVProf 5.5 [12] to measure the metrics and the overall execution time of the kernels. We evaluate benchmark performance on K20c and compare CDP overhead across all the three GPUs.

We compare the non-CDP implementation and the CDP implementation of the unstructured applications. In both cases, inputs are evenly partitioned among threads and CTAs. In the non-CDP case, DFP is handled by individual threads respectively, generally through loops. The CDP implementation uses dynamically launched kernels for parallel computations detected through DFP.

To evaluate the impact of CDP implementation on both the control flow and memory access, we use the following hardware metrics.

warp_execution_efficiency (WEE). This metric measures the ratio of active threads within a warp for all executed instructions (at warp level). It is an indication of the control divergence or workload unbalance in the unstructured applications. Note that NVProf does not allow separate metrics measurement for parent kernels and child kernels in CDP, so the metric measured by NVProf are affected by the parent kernel execution, the child kernel execution and the child *kernel launching overhead* (recall that the overhead includes child kernel parameter passing and device runtime management). We also propose an approach to measure and compute the warp execution efficiency excluding CDP kernel launching overhead and refer it as the ideal WEE (**WEEI**) since it represents the ideal efficiency that can be achieved:

$$\text{WEEI} = \frac{\text{WEE\_parent} * \text{inst\_parent} + \text{WEE\_children} * \text{inst\_children}}{\text{inst\_parent} + \text{inst\_children}}$$

In the equation, inst_parent and inst_children are the effective executed instruction by the parent and children respectively. When we measure WEE_parent and inst_parent, we remove the child kernel launch code from the parent kernel. A warm up kernel is executed before the parent kernel to make sure the parent kernel execution path does not change when child kernels are removed. When we measure WEE_children and inst_children, we extract the child kernels and launch them from the hosts using the same configuration and input data as the device launch. By doing this we are able to exclude the CDP *kernel launching overhead* in the measurement. We argue that this approach is accurate enough to generate WEEI as it only depends on execution path but not any hardware-dependent factors such as warp scheduling and child kernel scheduling policy. The purpose of WEEI is to demonstrate the potential benefit of CDP implementation of DFP by setting up the possible upper boundary.

ldst_replay_overhead (LSRP). This metric measures the average number of replays for each load/store instruction executed. Instructions are replayed when there is bank conflict or non-coalesced memory access. Therefore, LSRP is able to capture the memory irregularity [13]. Since LSRP may be dependent of the execution history (e.g. cache and RAM access history), it is not reasonable to separate the parent kernel and child kernels to measure ideal LSRP for CDP implementation as for WEE. However, we measure the number of load/store instruction separately and conclude that the load/store instructions from *kernel launching overhead* only comprise very small portion of all the load/store instructions, so the directly measured LSRP can still be a good indication for memory divergence affected by CDP.

l2_cache_hit_rate (L2HIT). This metric measures the L2 cache hit rate and captures the memory locality in the program either for a thread or for threads from interleaved warps. Again, we measure L2HIT directly without excluding the child *kernel launching overhead*.

B. Benchmarks

The following is a brief description of the unstructured applications used in our experiments.

Adaptive Mesh Refinement (AMR): AMR operates on grid-like structure and refine each cell in the grid according to certain conditions. We use AMR to represent the combustion simulation problem [14] where each cell in the grid is given an average temperature. Cells are refined to smaller cells according to the energy computed out of the average temperature. The process stopped until the energy of each cell in the grid is below a threshold. Non-CDP implementation uses one kernel for each refine level and cell refinement is perfomed by each individual thread with loop iterations. CDP implementation launches cell refinement kernel recursively when energy threshold condition is satisfied.

Barnes Hut Tree (BHT): The BHT problem is part of the Barnes-Hut NBody Simulation [15] which computes the forces between the points in the space. A tree is built where each leaf node only contains at most one data point. Each thread takes one data point and compute the force between that point and any other point if they are close or the center of mass of any other cell if they are far away. The algorithm needs each thread to traverse the tree depending on the point-to-point distance. CDP implementation launches a new kernel if one thread needs another level of tree traversal. The input to BHT are randomly generated data points.

Breadth-First Search (BFS): BFS algorithm searches and visits all vertices in a graph using breath-first patterns. Graph is stored in the CSR format where a column buffer stores all the edges that are connected to each source vertex and a row buffer stores the starting edge index of each vertex in the column buffer. The vertex frontier is maintained for each search iteration. Each thread takes one vertex in the frontier, expands the edges, marks visit information and puts unvisited vertices to the new frontier. CDP implementation launches child kernels dynamically to expand the vertices in parallel according to the vertex degree (number of edges connected to the vertex). We use three different graphs as the input to BFS [16]: citation network (citation), USA road network (usa_road) and a sparse matrix from Florida Sparse Matrix Collection (cage15).

Graph Coloring (CLR): Graph Coloring problem is widely used in lots of research domains, e.g. compiler register allocations. The goal is to assign a color to each vertex in the graph such that no neighboring vertices have the same color. The algorithm [17] starts by assigning each vertex with a random integer and then in each iteration, each thread takes a vertex and marks itself with the iteration color if its value is larger than any adjacent vertex. The vertices with color assigned are ignored in subsequent iterations. Similar as BFS, CDP implementation launches new child kernels to examine the neighbors for each vertex. The input to CLR are the three graphs used in BFS.

Regular Expression Match (REGX): Regular expression match is the centric for many search and pattern match problems, e.g. network packet routing. The regular expression pattern is represented by finite automata (FA) and stored in the memory as a graph where each vertex represents a state and the edges represent transitions between states [18]. Each thread takes an input stream and traverse the FA. If a match is found, the corresponding state in the FA is returned. CDP implementation recursively traverses the FA and examine the transition edges in parallel. The input to REGX are the DARPA network packets collection (regx_darpa) [19] and random string collection (regx_string).

Product Recommendation (PRE): Product recommendation systems are widely used in industry especially on e-commerce website. These systems predict the customer purchase behavior according to past purchase records. We focus on the item-based collaborative filtering algorithm for recommendation systems. One important part of item-based collaborative filtering is to construct an $M \times M$ similarity matrix of the items by examining the M items purchased by N customers. Each thread examines one customer and records item-item pairs into the similarity matrix [20]. For P items purchased by one customer, there are $P \times (P - 1)$ pairs to be recorded. CDP implementation launches child kernels to record these pairs in parallel. The input to PRE are data from MovieLens [21].

Relational Join (JOIN): The JOIN operator is relational algebra operator that is commonly used in relational database computation. We experiment on the inner JOIN algorithm where two input relation arrays are examined to generate a new relation array consisting of the key-value pairs where the keys are present in both of the input arrays. Each thread takes one element from one of the input arrays and uses binary search to find matching keys from the other array [22]. Workload imbalance can happen when matched element count varies for threads. CDP implementation resolves the problem by launching a new kernel to gather the result elements in parallel for each thread. The input data to JOIN are synthetic data arrays that have uniform distribution (join_uniform) and gaussian distribution (join_gaussian).

Single-Source Shortest Path (SSSP): SSSP is a classic graph problem which finds the paths with the minimal cost (sum of weights) from a given source vertex to all the vertices in the graph. Except the source vertex, all vertices are starting from infinite cost. It is then updated by examining the neighbors in each iteration to find the one with minimum cost after adding the weight from that neighbor to the vertex. CDP implementation launches a new kernel to examine the neighbors and uses reduction to find the minimum. Input to SSSP are the three graphs used in BFS.

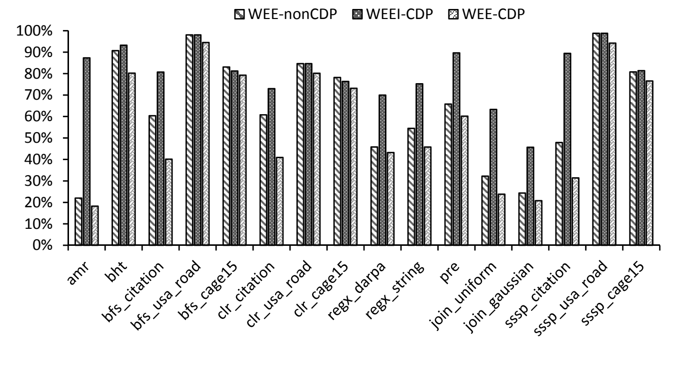

Fig. 1. Warp Execution Efficiency for non-CDP and CDP implementations.

VI. Evaluation and Analysis

In this section we report a comprehensive evaluation and analysis of the benchmark performance. We first compare the CDP and non-CDP implementations of the benchmarks in control flow behavior, memory behavior and overall execution time to illustrate the potential impact of handling DFP with CDP on the unstructured applications. We use different inputs to the benchmarks to capture behavior and performance on various characteristics. Then we evaluate CDP overhead in several aspects including kernel launch, memory footprint and algorithm overhead. We also analyze the child kernel workload intensity and scheduling policy. Finally we discuss the lessons learned from the observations.

A. Control Flow Behavior

We show WEE of non-CDP implementations, both WEE and WEEI of CDP implementations in Fig. 1. Recall that WEE-CDP includes the child *kernel launching overhead*.

WEE of the non-CDP implementations ranges from 21.9% to 98.8%. Low WEE indicates lower SIMD lane utilization or more workload imbalance in unstructured applications. By using CDP implementation for DFP, the workload imbalance or control divergence can be reduced and WEE can be effectively increased. We show WEEI which is the ideal WEE that can be achieved by applying CDP implementation excluding the *kernel launching overhead*. For most applications, WEEI increases 2.2% to 65.3% from WEE-nonCDP. Examples of such applications include AMR, BFS_citation, CLR_citation, SSSP_citation, PR, REGX and JOIN, all operating on highly irregular data structure or generating highly irregular computation. The refinement of AMR is completely dependent on the data point values and varies at large degree from one thread to another. For the citation network graph, vertex degree which represents the number of cited authors is largely different from each other. The potential benefit is substantial when using CDP implementations for these benchmarks.

On the other hand, some applications are showing no or negative potential improvement. For example, BFS, CLR and SSSP with USA road network do not achieve any WEEI increase at all. The reason is that the degree of vertices in USA road network graph generally ranges from one to four, which does not trigger the condition to launch a new child kernel (recall that at least 32 threads are needed in a child kernel). These benchmarks already have high WEE because of the relatively balanced workload among threads and CDP implementations

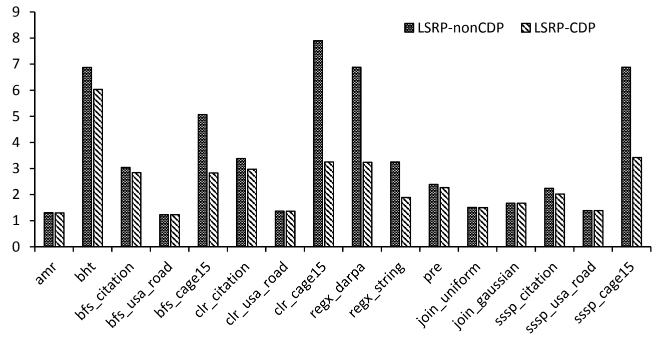

Fig. 2. Average number of load/store instructions replay.

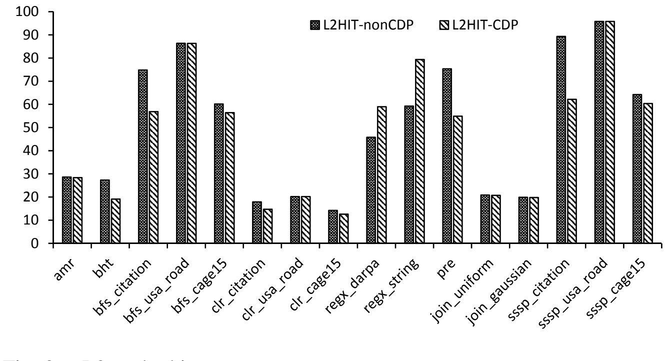

Fig. 3. L2 cache hit rate.

would not be necessary. It is even more interesting to notice that BFS and CLR for graph cage15 have WEEI decreased from WEE-nonCDP. The reason is that cage15 have relatively small variance in vertex degree. Launching a child kernel for some vertices but not for others actually intensify the workload imbalance problem, which result in decrease of WEE.

WEE-CDP shows the real measurement of WEE for CDP implementation. When including the *kernel launching overhead*, the SIMD lane efficiency decreases dramatically. Compared to WEE-nonCDP, WEE-CDP decreases from 3.5% to 20.2%. One hypothesis is *kernel launching overhead* introduces a large number of instructions with very low SIMD lane utilization and bring down the overall WEE (see kernel launching time analysis in section VI-D1). The more child kernels are launched to increase WEEI, the more overhead is introduced and the larger dropdown can be observed from WEEI to WEE-CDP.

Insight. For unstructured applications that exhibit severe workload imbalance and relatively high dynamic parallelism, CDP can potentially reduce control flow divergence. However, for applications like BFS_cage15 and CLR_cage15 that do not have high thread-level workload variance, CDP does not show performance advantages. We can envision a strategy for invoking CDP based on the degree of workload variance.

B. Memory Behavior

Fig. 2 shows LSRP for non-CDP implementation and CDP implementation to interpret the memory access irregularity. For all the benchmarks, CDP implementations reduce LSRP up to 58.8%. BFS, CLR and SSSP for cage15 and REGX have the most significant LDPR decrease among all benchmarks. These benchmarks have more scattered memory access by each thread within a warp in the non-CDP implementations. For example, the graph cage15 have distributed neighbor list so the vertex expansion from different threads access vertices far away from each other, generating many memory transactions. By using CDP to handle DFP in unstructured applications, threads in the child kernel executing the same memory instruction are more likely to access contiguous addresses. Memory divergence can be greatly reduced for these benchmarks since more coalesced memory accesses are generated.

The graph citation network, road network and PRE, on the other hand, does not show much change in LSRP. They have the characteristic that neighbor vertices are stored close to each other in the memory (Citations tends to be from the same list of authors for a research area, nearby cities are connected together in the road network and for the PRE system, people are more likely to choose similar items), so even the original non-CDP implementation does not exhibit much memory access irregularity. CDP implementations do not show much benefit in these cases.

The LSRP behaviors for AMR and JOIN have different explanation. Unlike other benchmarks that traverse some irregular data structure which may generate non-coalesced memory accesses, AMR and JOIN have irregular data computation procedure by following dynamic and data-dependent execution paths rather than irregular memory access patterns. They would also exhibit low memory divergence and not take advantage from CDP implementations in terms of LSRP.

We also measure the L2 cache hit rate with the metric L2HIT as shown in Fig. 3. Most of the benchmarks show unchanged or decreased L2 cache hit rate due to the fact that CDP implementations break the spatial locality found in the non-CDP implementations where each thread may access contiguous address in different loop iterations.

The exceptions are two REGX benchmarks which show 13.2% and 20.1% cache hit rate increase respectively. Considering LSRP are also increased, their behaviors demonstrate the CDP implementation reserve both spatial locality within a thread and across the intra-warp threads. In these cases the child kernels with close memory address accesses are scheduled together, thereby increasing the cache hit rate.

Insight. Depending on the data arrangement and access patterns, CDP may reduce memory divergence by generating more coalesced memory accesses. However, it could reduce cache hit rate since accesses that were serialized in time in a non-CDP implementation now are redistributed across child kernels that execute concurrently. We believe that this can be mitigated by sophisticated child kernel scheduling policies much for the same reasons interleaved warp scheduling is effective at hiding memory latency.

C. Overall Performance

To evaluate the overall performance of the benchmarks using CDP implementation, we measure the execution time of the computation kernels of the applications. Note that data transfer time between CPU and GPU are excluded. We also propose an approach to measure the ideal CDP implementation time excluding the *kernel launching overhead*. First, we replace each child kernel with a dummy kernel that has an empty function body and measure the overall execution time $t1$. Then we remove all the child kernel launch and measure

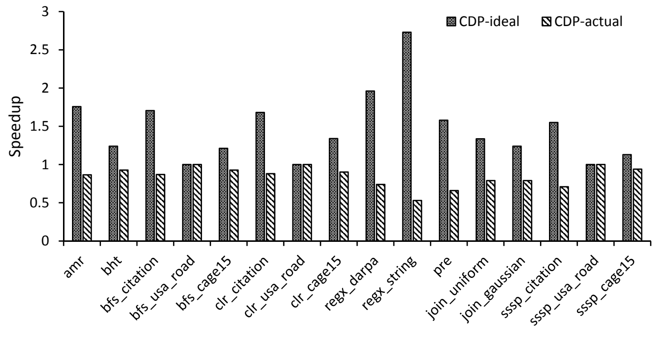

Fig. 4. Speedup of CDP implementations (ideal and measured) of unstructured applications over non-CDP implementations.

the execution time $t2$. In both cases, a warm up kernel is executed before the parent kernel to fill in the result data so that the execution paths of the parent kernel do not change. We use $t1 - t2$ as the ideal child kernel launching time and exclude it from the actual CDP implementation execution time to generate the ideal execution time as a lower boundary. Note the $t1 - t2$ depends on the number of child kernels which is determined by the patterns of dynamic parallelism in each application. While we choose the most straightforward CDP implementation without explicitly controling the number of child kernels, more sophisticated implementations are possible and could potentially reduce the overhead. The detailed algorithm design is out of the scope of this paper.

The speedup of both CDP ideal and actual execution time over nonCDP implementations are shown in Fig. 4. BFS, CLR and SSSP with USA road network input shows no speedup or slow down for both scenarios because the child kernels launching threshold is never satisfied and no child kernel is launched. Other benchmarks show 1.13x-2.73x speedup for CDP-ideal. REGX_darpa and REGX_string have highest ideal speedup 1.96x and 2.73x respectively, which can be justified through the observation that CDP implementations have both positive impact on WEEI and LSRP. However, when including the *kernel launching overhead*, no benchmark can perform better than the non-CDP implementation with an average of 1.21x slow down. An interesting fact is that the higher speedup of CDP-ideal over nonCDP, the more slowdown of CDP-actual over nonCDP, since applications that can take more advantage from CDP implementations have more child kernel launching and incur more overhead.

Insight. As the CDP implementations manage to reduce both control flow and memory access irregularity which are two essential metrics that affect the performance on GPU, execution speed up is expected. However, with CDP support on GPUs in its current form, the overhead of device-side kernel launches have a substantially negative influence on the overall performance negating those gains.

D. CDP Overhead

As discussed in the previous sections, CDP implementations introduce substantial overhead which may negate potential performance benefit brought by handling DFP using device-side kernel launches. We characterize such overhead in different aspects to get a comprehensive understanding of CDP.

1) CDP Launching Time: We measure the *CDP launching time* using $t1 - t2$ with different thread number in the parent kernel to control the total number of child kernel launches. We check the output PTX code and make sure the dummy child kernels are not eliminated by the compiler optimization. Recall that the *CDP launching time* includes time spent on kernel parameter parsing, calling `cudaGetParameterBuffer` and `cudaLaunchDevice`, as well as the time for device runtime to setup, enqueue and dispatch the child kernels. The time spent on data dumping by the parent kernel to pass data to the child kernels is excluded from *CDP launching time*.

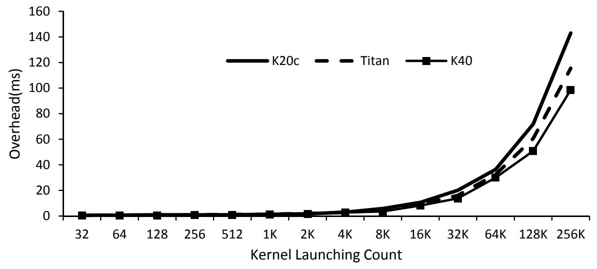

Fig. 5. CDP launching time.

Fig. 5 shows the result for different child kernel count across three different GPU platforms. For all three GPUs, the *CDP launching time* stay around 1ms for kernel launching count from 32 to 512. Then it scales with the kernel launch count and reaches 143ms, 115.5ms and 98.57ms for 256K child kernel launches on K20c, Titan and K40 respectively (in comparison, the execution time of a typical kernel in the nonCDP implementation of BFS_citation is 3.27ms). We use the same method to measure the *kernel launching time* for each benchmark and compute the ratio over the overall execution time in Fig. 4 which has average value of 36.1% and max value of 80.6%. The common problem for CDP implementation of the unstructured applications is that they require a large number of child kernel launch but the computation workload in each child kernel is very light. As the launching time scales with the number of child kernels, the performance can dramatically degrade.

2) Memory footprint: When using CDP, global memory in GPUs may be reserved by device runtime for child kernel launch. Device runtime maintains a kernel launching pool for all the launched but pending execution kernels due to unresovled dependency or lack of resources. The size of this pool is referred as *pending launch count limit*. For every pending launched child kernel, the device runtime uses reserved memory to store the launching information such as the parameters and configurations. The *pending launch count limit* can be specified using `cudaDeviceSetLimit` with the option `cudaLimitDevRuntimePendingLaunchCount`. CDP execution reports a runtime error if the number of kernels pending execution on the fly exceeds this limit. On the other hand, CDP allows parent kernels and child kernels to explicitly synchronize with each other by calling `cudaDeviceSyncrhonize`. The device runtime has to save the states of parent kernels when they are suspended and yield to the child kernels at the synchronization point. The reserved memory size depends on the synchronization depth which can be specified using `cudaDeviceSetLimit` with the option `cudaLimitDevRuntimeSyncDepth`. Again, CDP execution reports an runtime error if the actual synchronization depth exceeds the limit.

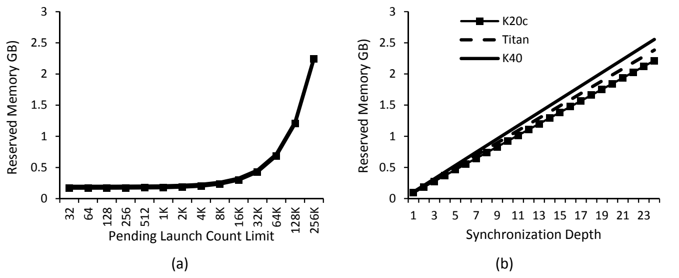

Fig. 6. Reserved global memory for CDP kernel launch and synchronization.

We measure the reserved memory size by calling the runtime API `cudaMemGetInfo` before and after `cudaDeviceSetLimit` and compute the free memory size difference. Fig. 6 shows the memory footprint for both scenarios.

As shown in Fig. 6(a), the memory size reserved stays the same for *pending launch count limit* less than 32, which are 172MB, 186MB and 202MB for K20c, Titan and K40 respectively. If the *pending launch count limit* does not exceed 32K, the memory reserved is less than 10% of the total global memory on K20c. We show the average minimum *pending launch count limit* required to execute each benchmark and the reserved memory size in Fig. 7. Again, the benchmarks show diverse behaviors. REGX_string requires 127K *pending launch count limit* and 1.2GB reserved memory. As discussed before, CDP implementation of REGX can greatly increase WEEI and decrease LSRP by launching many child kernels. As a tradeoff, it requires much more memory space reserved. To the extreme opposite, the graph USA road network requires zero *pending launch count limit* since the parallelism degree is very low in DFP and CDP is not activated. However, there are still 172MB reserved memory which is the minimum cost to pay to link against device runtime library with CDP functionality enabled.

Fig. 6(b) shows that the memory reserved for synchronization scales linearly with the synchronization depth. The highest synchronization depth 24 requires 2.2GB global memory reservation on K20c which is 44% of the total available GPU memory. A close analysis at the measurement shows that for each increased synchronization depth, the memory size reserved are 95MB, 102MB and 109MB for K20c, Titan and K40 respectively, which scale with the number of SMXs in each GPU. This is because when the parent kernels are suspended, all the data (including local, shared memory data and etc.) currently occupying each SMX should be saved. As the three GPUs have the same SMX architecture, the total amount of reserved memory should be that of each SMX multiply by the SMX count.

3) Algorithm overhead: Besides the overhead caused by the device runtime, sometimes the algorithm itself has to be changed for CDP implementation and may introduce overhead. Share memory usage in parent kernel can be tricky as the only ways child kernel can access the data is either through child kernel parameters or expensive global memory bypass. Therefore, algorithm has to be adapted to reduce shared memory passing between the parent and the children.

Spatial ordering requirement is another source that may introduce overhead. For example, in the JOIN benchmark, each block uses prefix-sum to compute the output offset for the result data since JOIN requires them to be strictly ordered.

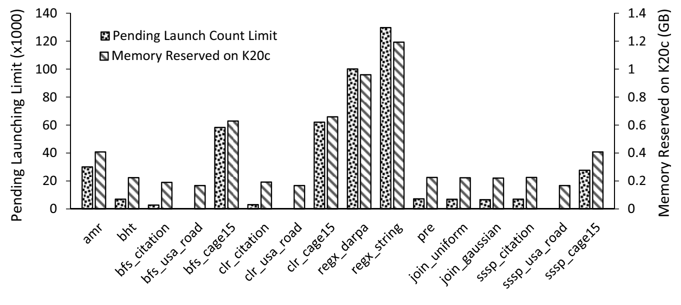

Fig. 7. Pending launch count limit and reserved memory size.

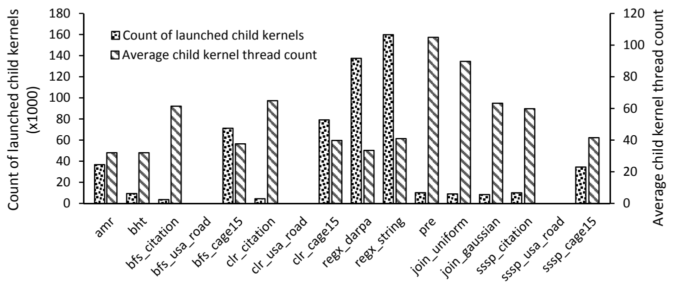

Fig. 8. Total child kernel launching count and their average thread count.

While in the CDP implementation, two prefix-sum are required instead of only one in the non-CDP implementation. One is used before the child kernel launch to compute the offset for child kernel output data, and the other one is required after the child kernel launch to compute the offset for remaining data that are not generated by the child kernel.

Insight. CDP introduces multiple sources of overhead from algorithm to device runtime management. The memory footprint reduces available global memory which can be a critical problem for large HPC applications. Both the *kernel launching overhead* and memory footprint scale with the number of launched child kernels. To reduce the overhead requires either the programmers to decrease child kernel launch count by developing more performance-aware algorithms for CDP implementation, or the GPU architecture and software stack to advance the technology for reducing the time and space overhead of device-side kernel launching.

E. Child Kernel Workload Intensity and Scheduling

We investigate the child kernel launching traces generated by NVProf to understand the child kernel workload intensity and scheduling. We first count the maximum number of child kernels launched by a parent kernel and the average thread number in these child kernels in each benchmark shown in Fig. 8. It can be noted that while a very large amount of child kernels are launched (up to 156K as in REGX_string), they are generally fine-grained kernels that perform very light workload (average kernel thread count is 44). Also note that the number of child kernels launched is only slightly larger (average 1.3x) than *pending launch count limit* shown in Fig. 7, which implies that most child kernels are launched together in a short period of time to quickly fill the launching pool.

Then we study the time stamp of the parent and child kernels for one iteration in BFS_citation benchmark and show the result in Fig. 9. Each vertical line in the figure marks

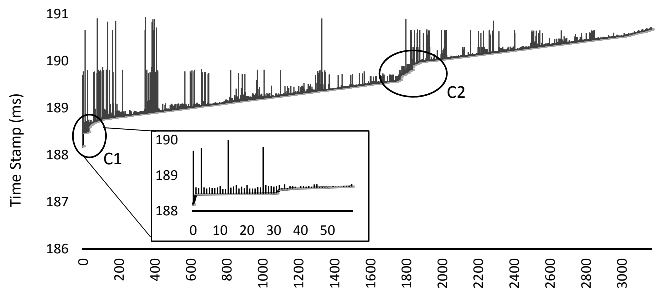

Fig. 9. Kernel execution trace for BFS_citation iteration 5.

the start and end execution time of a kernel. The first line is for the parent kernel and the remainings are for the child kernels. The general trend shown by the figure is that kernels are scheduled and executed with increasing time stamps until completion, which conform the fact that child kernels are launched when resources are available. There are two stages shown in the figure that present dramatic increase in time stamp, denoted by C1 and C2. C1 marks the early stage of the application, when the parent kernel starts launching several child kernels. A close look at C1 shows that 1) child kernels start execution before parent kernel is finished and 2) child kernels are executed concurrently (31 child kernels start execution at the same time). C2 marks the stage when the CTAs in the parent kernel start processing a new portion of the input vertices and generate a new round of child kernel launches. It shows that previous round of child kernel launches are gradually completed followed by the concurrent execution of newly launched kernels.

Insight. We find that using CDP often leads to more fine grained kernels compared to the host-side launched kernels, i.e., CDP implementations can generate a large number of child kernel launches, where often each kernel is relatively fine grained. This makes performance more sensitive to kernel level concurrency, kernel level scheduling policies, and kernel launching overhead. Alternatively, application developers may wish to cognizant of and sensitive to kernel level granularity when making nested kernel calls on the GPU.

F. Discussion

The experiments and measurements show that ideally dynamic device-side kernel launching could be a solution for irregular applications with dynamic structured parallelism. The current support of CDP in GPUs provides some preliminary perspective into this problem by showing potential performance benefit with non-trivial overhead. We identify and discuss here several aspects in software and hardware that might reduce the negative impact of CDP.

1) CDP programming methodology: As shown in the experiments, when using CDP to implement DFP in unstructured applications, the overhead scales with the number of child kernels launched and can negate potential performance benefits. One possibility is to merge multiple nested kernel to reduce the number of device-side launched kernels. For example, using one kernel to expand all the neighbors of the vertices processed by a CTA instead of limiting vertex expansion to only one thread. Such approaches rely on more sophisticated algorithms but then can retain both productivity and performance advantages.

2) Reduce reserved memory size: The memory footprint for a CDP kernel launch reduces free global memory that can be used by applications. We observe that child kernels produced by many workloads are often very similar to each other. Identical information stored for pending kernels (e.g. thread configuration for kernels) can be shared. Programmers or the compiler can provide hints to enable the device runtime manager to reduce the reserved memory size.

3) Revise kernel scheduling policy: More flexible scheduling policy can be proposed to increase child kernel execution efficiency. CDP implementations involve many fine-grained child kernels. The computation workload of these light-weight kernels are comparable to warps. Since child kernels are more likely to be launched together in a relatively short period as discussed before, there will be a relatively larger pool for kernels to be scheduled. Independent child kernel execution can be out-of-order to maximize the efficiency.

On the other hand, the current form of CDP complies with the same constraint on concurrent kernel execution for host-launched kernels, i.e., maximum 32 concurrent kernels can be executed on a GK110, while a maximum of 52 warps (4 warps scheduler each SMX $\times$ 13 SMX in K20c) can be scheduled concurrently. For a large number of child kernels with very few warps each, the GPU utilization could be increased if the concurrent kernel execution limit is higher or kernels are scheduled and interleaved similar to warp scheduling to hide memory latency.

VII. Related Work

Characterization and analysis of GPU applications can be traced to a very early time. Kerr et al. [23] characterized PTX kernels using different metrics and proposed recommendation methodology to write good GPU programs. Rodinia [24] proposed by Che et al., Parboil [25] proposed by Stratton et al. and SHOC[26] proposed by Danalis are some representative benchmarks used in GPU study. Their focus are more in the regular applications that can utilize the structured BSP model efficiently. Recently people have been investigating the performance of irregular applications on GPUs. Burtscher et al. [13] studied the behavior of irregular applications on GPUs with two quantitative metrics for control flow irregularity and memory access irregularity. Che et al. [27] used the Pannotia benchmark suite to illustrate the characteristics of irregular graph applications on GPUs. Our work, on the other hand, focus on the dynamic formed structured parallelism found in the unstructured applications. The applications that we investigate also include some new GPU research areas such as database and recommendation systems.

Researchers have been developing implementation using regular BSP model for unstructured applications. Merrill [28] implements BFS on GPUs using one CTA or one warp to explore adaptively the neighbors of one vertex in parallel which can utilize the SIMD lanes more efficiently. However, such effort has not been applied to other unstructured applications. We propose a different and general implementation strategy by using CDP which can be more flexible as the dynamic workload are not restricted by the CTA size of the parent kernels.

As a newly introduced technology, CDP has not been widely used for GPU applications. Only a few relevant researches have been reported. Wang et al. [29] proposed the CDP implementation of graph-based substructure pattern mining. DiMarco et al. [30] analyzed the clustering algorithm with CDP implementation. Our work applies CDP into various benchmarks and report a more comprehensive analysis. Yang et al. [31] proposed a compiler technique to handle nested parallelism in the GPU applications based on the observation that CDP dramatically reduces memory bandwidth. They also analyzed the nested parallelism in several benchmarks. While we agree that it is diffcult to fully utilize the current form of CDP to improve performance due to its non-trivial overhead, we target for a more thorough understanding of CDP and provide insights into both the advantages and disadvantages of using CDP particularly for unstructured applications.

VIII. Conclusion

In this paper we study the dynamically formed structured data parallelism in unstructured applications and implement them with the new CUDA Dynamic Parallelism technique on GPUs. We use a set of metrics to evaluate and analyze the potential performance benefit of the CDP implementations on the control flow behavior and memory behavior on several unstructured benchmark applications. We also present a comprehensive understanding of the efficiency of CDP in terms of runtime, memory footprint and algorithm overhead. The experiments show that CDP implementation can achieve 1.13x-2.73x potential speedup but the huge kernel launching overhead could negate the performance benefit. We provide the insights including the recommended programming methodology and possible architectural or software stack revisions to increase the kernel launching and scheduling efficiency.

References

[1] J. A. Anderson, C. D. Lorenz, and A. Travesset, "General purpose molecular dynamics simulations fully implemented on graphics processing units," *Journal of Computational Physics*, vol. 227, no. 10, pp. 5342–5359, 2008.

[2] J. Mosegaard and T. S. Sørensen, "Real-time deformation of detailed geometry based on mappings to a less detailed physical simulation on the gpu," in *Proceedings of the 11th Eurographics conference on Virtual Environments*. Eurographics Association, 2005, pp. 105–111.

[3] V. Podlozhnyuk, "Black-scholes option pricing," 2007.

[4] S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison *et al.*, "Optix: a general purpose ray tracing engine," in *ACM Transactions on Graphics (TOG)*, vol. 29, no. 4. ACM, 2010, p. 66.

[5] Khronos, "The opencl specification version 2.0," 2014.

[6] NVIDIA, "Nvidia's next generation cuda compute architecture: Kepler gk110," 2012.

[7] ——, "Cuda c programming guide version 5.5," 2013.

[8] W. W. 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*. IEEE Computer Society, 2007, pp. 407–420.

[9] J. Meng, D. Tarjan, and K. Skadron, "Dynamic warp subdivision for integrated branch and memory divergence tolerance," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3. ACM, 2010, pp. 235–246.

[10] NVIDIA, "Cuda dynamic parallelism programming guide," 2013.

[11] Y. Saad, *SPARSKIT: A basic toolkit for sparse matrix computations*. Research Institute for Advanced Computer Science, NASA Ames Research Center Moffet Field, California, 1990.

[12] NVIDIA, "Cuda profiler user's guide version 5.5," 2013.

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

[14] A. Kuhl, "Thermodynamic states in explosion fields," 2010.

[15] M. Burtscher and K. Pingali, "An efficient cu da implementation of the tree-based barnes hut n-body algorithm," *GPU computing Gems Emerald edition*, p. 75, 2011.

[16] D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, "10th dimacs implementation challenge: Graph partitioning and graph clustering, 2011."

[17] P. C. Jonathan Cohen, "Efficient graph matching and coloring on the gpu," 2012.

[18] L. Wang, S. Chen, Y. Tang, and J. Su, "Gregex: Gpu based high speed regular expression matching engine," in *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2011 Fifth International Conference on*. IEEE, 2011, pp. 366–370.

[19] J. McHugh, "Testing intrusion detection systems: a critique of the 1998 and 1999 darpa intrusion detection system evaluations as performed by lincoln laboratory," *ACM transactions on Information and system Security*, vol. 3, no. 4, pp. 262–294, 2000.

[20] C. H. Nadungodage, Y. Xia, J. J. Lee, M. Lee, and C. S. Park, "Gpu accelerated item-based collaborative filtering for big-data applications," in *Big Data, 2013 IEEE International Conference on*. IEEE, 2013, pp. 175–180.

[21] "Movielens," http://movielens.umn.edu.

[22] G. Diamos, H. Wu, J. Wang, L. A., and Y. S., "Relational algorithms for multi-bulk-synchronous processors," in *18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP13)*, February 2013.

[23] A. Kerr, G. Diamos, and S. Yalamanchili, "A characterization and analysis of ptx kernels," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. IEEE, 2009, pp. 3–12.

[24] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron, "A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads," in *Workload Characterization (IISWC), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 1–11.

[25] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," *Center for Reliable and High-Performance Computing*, 2012.

[26] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The scalable heterogeneous computing (shoc) benchmark suite," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. ACM, 2010, pp. 63–74.

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

[28] D. Merrill, M. Garland, and A. Grimshaw, "Scalable gpu graph traversal," in *ACM SIGPLAN Notices*, vol. 47, no. 8. ACM, 2012, pp. 117–128.

[29] F. Wang, J. Dong, and B. Yuan, "Graph-based substructure pattern mining using cuda dynamic parallelism," in *Intelligent Data Engineering and Automated Learning–IDEAL 2013*. Springer, 2013, pp. 342–349.

[30] J. DiMarco and M. Taufer, "Performance impact of dynamic parallelism on different clustering algorithms," in *SPIE Defense, Security, and Sensing*. International Society for Optics and Photonics, 2013, pp. 87 520E–87 520E.

[31] Y. Yang and H. Zhou, "Cuda-np: Realizing nested thread-level parallelism in gpgpu applications," 2014.