2015 44th International Conference on Parallel Processing

Nested Parallelism on GPU: Exploring Parallelization Templates for Irregular Loops and Recursive Computations

Da Li, Hancheng Wu, Michela Becchi
Dept. of Electrical and Computer Engineering
University of Missouri - Columbia
da.li@mail.missouri.edu, hancheng.wu@mail.missouri.edu, becchim@missouri.edu

***Abstract*—The effective deployment of applications exhibiting irregular nested parallelism on GPUs is still an open problem. A naïve mapping of irregular code onto the GPU hardware often leads to resource underutilization and, thereby, limited performance. In this work, we focus on two computational patterns exhibiting nested parallelism: irregular nested loops and parallel recursive computations. In particular, we focus on recursive algorithms operating on trees and graphs. We propose different parallelization templates aimed to increase the GPU utilization of these codes. Specifically, we investigate mechanisms to effectively distribute irregular work to streaming multiprocessors and GPU cores. Some of our parallelization templates rely on dynamic parallelism, a feature recently introduced by Nvidia in their Kepler GPUs and announced as part of the OpenCL 2.0 standard. We propose mechanisms to maximize the work performed by nested kernels and minimize the overhead due to their invocation.**

Our results show that the use of our parallelization templates on applications with irregular nested loops can lead to a 2-6x speedup over baseline GPU codes that do not include load balancing mechanisms. The use of nested parallelism-based parallelization templates on recursive tree traversal algorithms can lead to substantial speedups (up to 15-24x) over optimized CPU implementations. However, the benefits of nested parallelism are still unclear in the presence of recursive applications operating on graphs, especially when recursive code variants require expensive synchronization. In these cases, a flat parallelization of iterative versions of the considered algorithms may be preferable.

I. Introduction

In the last few years, the GPU architecture has increasingly become more general purpose. Nvidia GPUs, for example, have recently seen the inclusion of caches in their memory hierarchy, the progressive relaxation of their memory coalescing rules and the introduction of more efficient atomic and floating point operations. More recently, Nvidia has introduced the ability to perform nested kernel invocations in their Kepler GPUs. This feature, also known as *dynamic parallelism*, has been added to the OpenCL 2.0 standard and will soon be released in AMD GPUs as well.

These architectural and programmability enhancements have favored the adoption of GPUs not only as accelerators of regular applications (such as those operating on dense matrices and vectors), but also of irregular ones. Irregular applications, often operating on graph and tree data structures, are characterized by irregular and unpredictable memory access patterns, frequent control flow divergence and a degree of parallelism that is known only at runtime.

While dynamic parallelism has the potential for increasing the GPU programmability and allowing support for recursive algorithms, the effective use of this feature – both as a load balancing mechanism and as an enabler of recursive algorithms – must still be understood. First, it is important to determine which classes of applications may benefit from the use of dynamic parallelism rather than from a flat parallelization of the code. Second, dynamic parallelism in CUDA has fairly elaborated semantics, and the programmer is still left with the burden of understanding its memory and execution model.

In this work we focus on irregular applications exhibiting nested parallelism. Specifically, we consider two categories of computational patterns: parallelizable irregular nested loops and parallelizable recursive codes. In the former, the amount of work associated to the iterations of inner loops can vary across the iterations of their outer loops. In the latter, recursive calls may spawn different amount of parallel work. As a consequence of this uneven work distribution, simple parallelization templates handling all the loop iterations or recursive calls in the same way may lead to hardware underutilization. On the other hand, adding simple load balancing mechanisms to the parallelization may allow a better mapping of work to hardware and, consequently, it may improve the application performance. Here, we focus on simple load balancing schemes and study the use of dynamic parallelism as a means to achieve better hardware utilization. In particular, we design parallelization templates aimed to maximize the work performed by nested kernels and minimize their overhead. Our parallelization techniques can be incorporated in compilers, thus freeing the programmer from the need to worry about the mapping of work to the hardware and to understand the complex semantics of GPU dynamic parallelism.

In this work, we make the following contributions:

- We propose and study several parallelization templates to allow the effective deployment on GPU of iterative and recursive computational patterns that present uneven work distribution across iterations and recursive calls.
- We explore the use of nested parallelism on GPU in the context of two computational patterns: irregular nested loops and recursive computations.
- We evaluate the proposed parallelization templates and the use of dynamic parallelism on an irregular matrix computation (*SpMV*), four graph algorithms (*SSSP*, *betweenness centrality, PageRank* and *BFS*), and two tree traversal algorithms (*tree descendants* and *tree height*).

II. Parallelization Templates

A. Motivation

GPUs present two levels of hardware parallelism:

0190-3918/15 $31.00 © 2015 IEEE
DOI 10.1109/ICPP.2015.107

streaming multiprocessors and CUDA cores in Nvidia's parlance, and SIMD units and stream processors in AMD's. Commonly used programming models for many-core processors, such as CUDA and OpenCL, expose this two-level hardware parallelism to the programmer through a two-level multithreading model: at a coarse grain, CUDA thread-blocks and OpenCL work-groups are mapped onto streaming multiprocessors and SIMD units; at a fine grain, CUDA threads and OpenCL work-items are mapped onto cores and stream processors. An additional level of parallelism is provided by the opportunity to launch multiple kernels concurrently through CUDA streams and OpenCL command queues. In addition, Kepler GPUs allow nested kernel invocations; this functionality, called dynamic parallelism, has recently been announced as part of the OpenCL 2.0 standard and will soon be included in AMD devices. In the remainder of this paper, we will use CUDA terminology.

Despite the presence of hardware and software support for nested parallelism, finding the optimal mapping of algorithms exhibiting multiple levels of parallelism onto GPUs is not a trivial problem. Especially in the presence of irregular computation, a naïve mapping of irregular code onto the GPU hardware can lead to resource underutilization and, thereby, limited performance. Besides allowing recursion, dynamic parallelism has the potential for enabling load balancing and improving the hardware utilization. This is because nested kernels, each with a potentially different degree of parallelism, are dynamically mapped to the GPU cores by the hardware scheduler according to the runtime utilization of the hardware resources. Unfortunately, the effective use of this feature has yet to be understood: the invocation of nested kernels can incur significant overhead (due to parameter parsing, queueing, and scheduling) [1, 2] and may be beneficial only if the amount of work spawned is substantial.

In this work, we consider two categories of computational patterns involving nested parallelism: applications containing parallelizable irregular nested loops (Section II.B) and recursive algorithms (Section II.C). We propose and analyze different parallelization templates aimed to improve the hardware utilization and the performance – some of them leveraging the dynamic parallelism feature. Our goal is to propose and evaluate simple and yet effective mechanisms that can be incorporated in GPU compilers.

B. Irregular Nested Loops

The hierarchical nature of the GPU architecture leads to two logical ways to map parallel loops onto the hardware: different loop iterations may be mapped either to GPU cores (*thread-based mapping*) or to streaming multiprocessors (*block-based mapping*) [3]. In the former case, loop iterations are assigned to threads; in the latter, they are assigned to thread-blocks. For brevity, we will use the terms *thread-* or *block-mapped loops* and *kernels* to indicate loops and kernels parallelized with one of these two approaches. In the presence of loop nesting, the mapping of inner loops depends on the mapping performed on the outer loops. For example, thread-based mapping on an outer-loop will cause serialization of all inner loops, while block-based mapping

(a) irregular nested loop
```
foreach (int i = 1 to N)
  foreach (int j = 1 to f[i])
    work(i,j);
```
(b) dual-queue
```
set_low = {i :: f[i] ≤ lb_THRES}
set_high = {i :: f[i] > lb_THRES}
thread-mapped_kernel(set_low)
block-mapped_kernel(set_high)
```
(c) delayed-buffer
```
thread-mapped-loop(i){
  if(f[i] ≤ lb_THRES)
    for (int j = 1 to f[i]) work(i,j)
  else
    buffer.add(i);
}
blk-mapped-exec(buffer, work);
```
(d) dynamic parallelism œnaïve
```
thread-mapped-loop(i){
  if(f[i] ≤ lb_THRES)
    for (int j = 1 to f[i]) work(i,j)
  else
    blk-mapped_nested-kernel_TH(i,work);
}
```
(e) dynamic parallelism œoptimized
```
thread-mapped-loop(i){
  if(f[i] ≤ lb_THRES)
    for (int j = 1 to f[i]) work(i,j)
  else
    buffer_SM.add(i);
}
blk-mapped_nested-kernel_BL(buffer_SM,work);
```
Figure 1. Parallelization templates for irregular nested loops.

on an outer-loop will allow thread-based mapping on the inner loops. In addition, *stream-based mapping* (whereby different iterations of the loop are assigned to different CUDA streams) offers an additional degree of freedom to the parallelization process. During code generation, compiler analysis is required to identify the type of GPU memory where each variable must be stored and the need for synchronization and reduction operations to access shared variables. For example, in the presence of a block-mapped outer loop and a thread-mapped inner loop, variables defined inside the outer loop but outside the inner loop will be shared by all threads in a block. Therefore, these variables will need to be stored in shared memory, and their access by threads within the inner loop will require synchronization.

Simply relying on thread- and block-based mapping in the parallelization process is acceptable for *regular nested loops*, wherein the number of iterations of an inner loop does not vary across the iterations of the outer loop. However, this simple solution may lead to inefficiencies when applied to *irregular nested loops*, for which this property does not hold. Irregular nested loops have the structure of the code in Figure 1(a). As can be seen, the number of iterations of the inner loop is a function of *i*, the outer loop variable. In the case of irregular nested loops, the use of thread-based mapping on the outer-loop may cause warp divergence (i.e., different threads are assigned different amounts of work), while the use of block-based mapping will lead to uneven block utilization, which in turn may cause GPU underutilization. Load balancing of irregular nested loops is

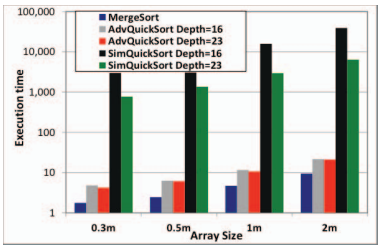


Figure 2: Execution time of sort algorithms (y-axis in $\log_{10}$ scale)

one of the use cases for GPU dynamic parallelism. By launching nested grids of variable size, dynamic parallelism has the potential for improving the GPU utilization. However, despite its overhead has been reduced in recent CUDA distributions, the effective use of this feature depends on the amount of work spawned in each kernel invocation.

We consider different parallelization templates aimed to perform load balancing in the presence of irregular nested loops. The proposed code variants trade off the advantages and disadvantages of thread- and block-based mapping, respectively. In particular, in Figure 1(b-e) we illustrate the load-balancing variants of the thread-based mapping template. Those variants rely on a load balancing threshold parameter (lb_{THRES}). The *dual-queue* template in Figure 1(b) divides the elements processed in the outer loop in two queues depending on the number of iterations they require in the inner-loop and processes those queues separately using thread- and block-based mapping, thus reducing the warp divergence of the former and the block-level work-imbalance of the latter. The *delayed-buffer* template in Figure 1(c) delays the execution of large iterations of the outer loop by queuing them in a buffer and then processing them using block-based mapping. We consider two versions of this template: one that stores the buffer in global memory (*dbuf-global*), thus requiring two kernel calls and allowing redistributing the work among thread-blocks in the second phase; the other that stores the buffer in shared memory (*dbuf-shared*), thus requiring a single kernel invocation but possibly leading to uneven work distribution among thread-blocks. The *naïve dynamic parallelism* (*dpar-naïve*) template in Figure 1(d) invokes a nested call for each "large" iteration (and performs the dynamic parallelism calls at a thread level). Finally, the *optimized dynamic parallelism template* (*dpar-opt*) in Figure 1(e) delays spawning nested calls to a second-phase; by invoking a single dynamic parallelism call for each thread-block, this code variant spawns fewer and larger kernels.

We note that GPU dynamic parallelism has fairly elaborate semantics. For example, nested kernel calls are performed by threads, but their synchronization happens at the level of thread-blocks; registers and shared memory variables are not visible to nested kernels; memory coherence and consistency between parent and child kernels require explicit synchronization; and concurrent execution requires the use of CUDA streams. Fortunately, the proposed parallelization templates for nested loops are simple and can be incorporated in a compiler, allowing the programmer to write only the simplified code in Figure 1(a). The automatic generation of the code variants corresponding to the proposed parallelization templates by a compiler will therefore hide from the programmer not only the two-level hardware and software organization of the GPU, but also the execution and memory model of GPU dynamic parallelism.

Several factors may impact the effectiveness of the proposed parallelization templates. For example, the optimal load balancing threshold (lb_{THRES}) will depend on the underlying dataset and algorithm. In addition, the performance of each parallelization template will depend on the characteristics of the algorithm (that is, the nature of the *work* in Figure 1). We explore these aspects in our experimental evaluation (Section III.B).

C. Recursive Functions

Recursive functions are a second use case for dynamic parallelism. The current CUDA distribution contains a few examples of recursive kernels, some of which, however, perform worse than equivalent, non-recursive codes. QuickSort is one of these examples. The CUDA SDK contains two implementations of QuickSort, both relying on dynamic parallelism: Simple QuickSort and Advanced QuickSort. We compared these two recursive codes with a non-recursive MergeSort kernel, also part of the CUDA SDK. Figure 2 shows the performance results obtained by running these sort implementations on arrays whose size varies from 300 thousand to 2 million elements. The performance of the recursive implementation depends on the setting of the *recursive depth* parameter, which allows trading off the load balancing effect of dynamic parallelism and its nested kernel call overhead. In fact, when the recursive depth limit is reached, Simple and Advanced QuickSort invoke a flat kernel (Selection and Bitonic Sort, respectively) to handle the remaining subarrays. As can be seen, while Advanced QuickSort outperforms Simple QuickSort, the best results are achieved by the non-recursive MergeSort kernel on all datasets. To conclude, while QuickSort is an interesting use case for dynamic parallelism, the overhead of the nested kernel calls and their need for synchronization makes a non-recursive GPU implementation preferable to a recursive one even in the presence of more optimized and complex code.

We aim to understand in which situations implementing recursive kernels using dynamic parallelism is preferable to non-recursive GPU variants of the algorithms considered. To be beneficial, the nested kernel invocations must spawn a significant amount of parallel work. Fortunately, parallel algorithms operating on trees and graphs often perform traversals over multiple paths or branches, and such traversals can be executed in parallel. These algorithms can therefore present a substantial amount of parallelism on large datasets. In addition, in the presence of irregular graphs and trees, the dynamic invocation of multiple, differently configured grids may lead to better load balancing of the parallel work and higher GPU utilization.

Again, our goal is to propose parallelization templates that can be incorporated in a compiler, so as to free the programmer from the need to write parallel recursive functions optimized for the GPU hardware hierarchy. In addition, we want to allow the generation of GPU code also for devices that do not support nested kernel invocations. To

this end, whenever possible, we extract an iterative version of the algorithm using mechanisms such as tail recursion elimination and autoropes [4] (the latter applicable to tree traversal algorithms). We then parallelize the iterative version of the code using thread- or block-based mapping (*flat parallelism*) and the recursive version of the code using different hardware mapping options.

As illustration, we consider the recursive computation of the number of descendants in a tree. Figure 3(a) shows a serial, recursive version of the algorithm. This code assumes that the *descendants* array, which stores the number of descendants of all nodes, is initialized to all 1s (that is, every node is a descendant of itself). The iterative version of the code in Figure 3(b) can be obtained by applying recursion elimination techniques. The flat kernel in Figure 3(c) parallelizes the iterative version of the code using thread-based mapping. We consider two parallel recursive code variants. The *naïve* version in Figure 3(d) parallelizes the recursive code over the children of the current node using thread-based mapping; each thread can spawn a parallel kernel consisting of a single thread-block. The *hierarchical* version in Figure 3(e) performs block-based parallelization of the recursive code over the children of the current node and thread-based parallelization over its grandchildren. The nested kernel invocations in this case happen at the thread-block level, and spawn hierarchical grids of threads. We perform recursion on the children of the current node. Recursion on the grandchildren would also be possible, but would cause more nested kernels to be spawned (thus degrading performance). Again, our goal is for the programmer to provide only the code in Figure 3(a): template-based parallelization can free the programmer from dealing with the semantics of nested parallelism for GPUs and the hierarchical hardware organization of these devices.

In Section III.C, we evaluate the proposed parallelization template on different recursive algorithms.

```
(a) recursive serial code
rec_function(graph g, node n){
  if (!leaf(n)){
    for (node c ∈ g.children(n)){
      rec_function(g,c);
      g.descendants[n]+=g.descendants[c];
} } }
(b) iterative serial code
iter_function(graph g){
  for (node n ∈ g.nodes)
    for (node p = g.parent[n]; p≠λ; p=g.parent[p])
      g.descendants[p]+=1;
}
(c) flat parallelism
flat_kernel(graph g){
  thread-mapped-loop(node n ∈ g.nodes){
    for (node p = g.parent[n]; p≠λ; p=g.parent[p])
      atomic{g.descendants[p]+=1;}
} }
(d) recursive parallelism – naïve approach
naive_rec_kernel(graph g, node n){
  thread-mapped-loop(node c ∈ g.children(n)){
    if (!leaf(c))
      naive_rec_kernel<1,block_SIZE>(g,c);
    atomic{g.descendants[n]+=g.descendants[c];}
}
(e) recursive parallelism – hierarchical approach
hier_rec_kernel(graph g, node n){
  block-mapped-loop(node c ∈ g.children(n)){
    bool recurse_SHMEM=false;
    if (!leaf(c)){
      thread-mapped-loop(node gc ∈ g.children(c)){
        if (!leaf(gc)) recurse_SHMEM=true;
      }
      if (recurse_SHMEM)
        hier_rec_kernel<grid_SIZE,block_SIZE>(g,c);
      else
        g.descendants[c]+=g.num_children(c);
    }
    atomic{g.descendants[n]+=g.descendants[c];}
} }
```

Figure 3. Application of different parallelization templates for recursive algorithms to the computation of tree descendants.

III. Experimental Evaluation

A. Experimental Setup

Hardware and Software Setup: We run all our experiments on a server equipped with a Xeon E5-2620 CPU and an Nvidia K20 GPU. The machine uses CentOS release 6.4. We compiled and run our code using gcc v4.4.7 and CUDA v6. We used Nvidia Visual Profiler to collect the profiling data presented in our analysis. Each data point is computed by averaging the kernel execution time reported over ten runs.

Benchmark Applications: We evaluated the performance of the proposed parallelization templates on seven applications falling in two categories: algorithms including irregular nested loops (*SSSP, BC, PageRank, SpMV*) and recursive applications (*tree descendants*, *tree heights* and *BFS*). Whenever available, we used open-source implementations of these applications as baselines.

- **Single-Source Shortest Path (*SSSP*)**: For *SSSP,* we used the thread-mapped implementation described in [5] as baseline. SSSP is a memory intensive application with scattered memory accesses.
- **Betweenness Centrality (*BC*)**: A node's *BC* is an indicator of its centrality in a network, and its value is equal to the number of shortest paths from all nodes to all others that pass through that node. Our parallel implementation is based on [6] and operates in two phases. First, it constructs the shortest paths tree using BFS (we consider unweighted graphs); second, it computes the BC value by traversing the shortest path tree. Both phases present irregular nested loops and scattered memory accesses.
- **PageRank**: *PageRank* is a popular graph analysis algorithm to rank web pages. We consider the GPU implementation described in [7] as a reference. This algorithm contains a parallelizable, irregular nested loop: each iteration of the outer loop processes a different webpage (node in a graph); the inner loop collects ranks from the neighbors of the considered node.
- **Sparse Matrix-Vector multiplication (*SpMV*)**: *SpMV* [8] calculates the product of a sparse matrix and a dense vector, and is an important building block for diverse applications in scientific computing, financial modeling and information retrieval. Since the sparse matrix is represented in Compressed Sparse Row format, the nested loop within the matrix multiplication algorithm is irregular.

- **Tree Descendants/Heights**: *Tree Descendants* and *Tree Heights* are tree traversal algorithms that calculate the number of descendants and the heights of all nodes in a tree. Figure 3 shows the pseudo-code of our flat and recursive implementations of tree traversal; we have generated similar code for tree height.
- **Breadth First Search (*BFS*)**: For *BFS*, we used the thread-mapped implementation described in [5] as code variant exhibiting flat parallelism. This implementation is work-efficient and traverses the graph level by level. Our recursive code recurses over the neighbors of each node. Since the scheduling of parallel recursive calls is non-deterministic, in our recursive code variants each node can be traversed multiple times provided that its level decreases at each traversal. In addition, since two nodes in a graph may have overlapping neighborhoods, the recursive implementations require atomic operations.

Datasets: For *SSSP*, *PageRank* and *SpMV*, we use *CiteSeer,* a paper citation network from the *DIMACS* implementation challenges [9]. *CiteSeer* has about 434 thousand nodes, 16 million edges and a node outdegree that varies from 1 to 1,188 across the graph (with an average of 73.9). For *BC* we use Wikipedia's who-votes-on-whom network (*Wiki-vote*) [10], a small-world network consisting of about 7 thousand nodes, 100 thousand edges and a node outdegree that varies from 0 to 893 across the graph (with an average of 14.6.9). For recursive algorithms, we use synthetic datasets as described in Section III.C.

B. Experiments on Irregular Nested Loops Algorithms

In this section, we first discuss the selection of the kernel configurations used in our experiments; we then illustrate the rationale of our experiments by presenting some results on *SSSP*; and we finally complete our discussion by comparing the results reported on *PageRank*, *BC* and *SpMV*.

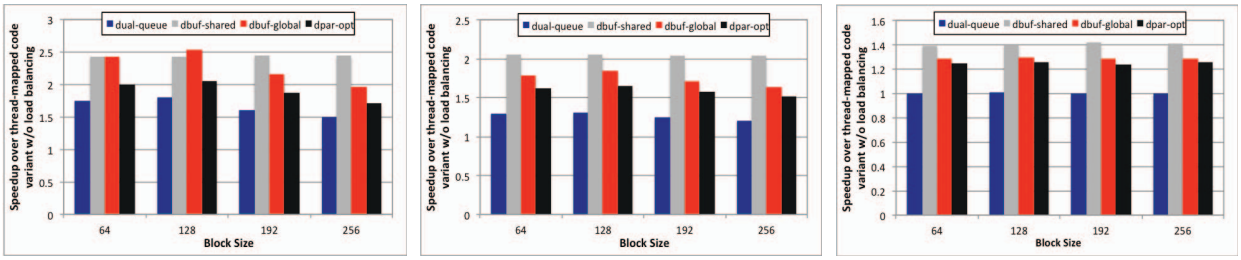
All the charts presented in this section report the speedup of the code variants derived from the use of the parallelization templates in Figure 1 over a baseline implementation that uses thread-mapping in the outer loop and no load balancing. The baseline GPU implementations achieve the following speedups over serial CPU code: 8.2x (*SSSP*), 2.5x (*BC*), 15.8x (*PageRank*) and 2.4x (*SpMV*).

Kernel configurations – Recall that the parallelization templates in Figure 1 include two phases: one using thread-based mapping and one using block-based mapping. For example, the *dbuf-global* scheme first invokes a kernel where the outer loop is parallelized with thread-based mapping, and then it invokes a kernel that uses block-based mapping to process the delayed buffer. Here, we discuss the selection of the kernel configuration used in both cases.

For thread-based mapping, we leverage the CUDA occupancy calculator to determine the optimal thread-block size. Since the considered applications have a low register and shared memory utilization (and some of them do not use shared memory at all), the optimal block size results to be large in all cases. Specifically, we use 192 threads per block, equaling the number of cores per streaming multiprocessor on Kepler GPUs. We recall that in a thread-mapped implementation each thread is assigned one or more iterations of the outer loop in Figure 1(a). Hence, we configure the number of blocks to be run based on the block size, the total number of iterations to be run and the maximum grid configuration allowed on Kepler GPUs.

In case of block-based mapping, each iteration of the outer loop is assigned to a thread-block, and threads within a block execute iterations of the inner loop. A small thread-block configuration will tend to assign multiple iterations of the inner loop to each thread. Conversely, large blocks may lead to hardware underutilization, as some iterations of the outer loop may not present enough parallelism to fully exploit the GPU cores on a streaming multiprocessor. We recall that the value of the lb_{THRES} (load balancing threshold) parameter determines the iterations of the outer loop that are processed in the second, block-mapped phase of the code.

We performed a set of experiments to determine good block size configurations to be used in the block-mapped portions of the code under different lb_{THRES} settings. Figure 4 shows the results of experiments performed on *SpMV* using different block size configurations and various settings of parameter lb_{THRES}. All experiments are run on the *CiteSeer* network. We omit the *dpar-naïve* implementation because, as we will show later, it greatly underperforms the other code variants. According to the CUDA occupancy calculator, small blocks consisting of 32 threads lead to low hardware occupancy. Our experiments confirmed low performance with fewer than 32 threads per block; therefore, here we consider block sizes ≥ 64. As can be seen, the performance is insensitive to the block size but mainly affected by lb_{THRES}. Some templates perform better in the presence of smaller blocks, especially for small values of lb_{THRES}. We observed



(a) lb_{THRES} = 64 (b) lb_{THRES} = 128 (c) lb_{THRES} = 192

Figure 4. *SpMV*: Speedup of load balancing code variants over basic thread-mapped implementation under different lb_{THRES} settings and varying block sizes. The input dataset is the sparse matrix representation of the *CiteSeer* network. In the baseline implementation the block size is set to 192. The *x*-axis shows the block sizes used by the portions of the code parallelized through block-based mapping. The naïve dynamic parallelism-based implementation (not shown for readability) is significantly slower than the other code variants.

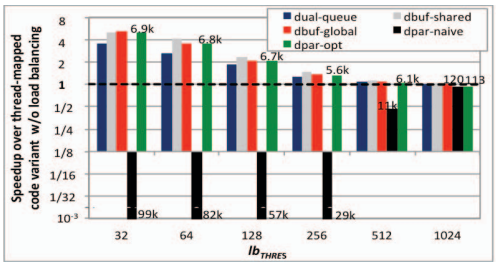


Figure 5. *SSSP*: Speedup of load balancing code variants over basic thread-mapped implementation. Experiments performed on the *CiteSeer* dataset and Nvidia K20 GPU. The numbers on the bars indicate the number of nested kernel calls performed by dynamic parallelism-based solutions.

similar results on the other applications with different datasets. These results can be explained as follows. A block size larger than the value of lb_{THRES} may lead to hardware underutilization. To understand why, refer to the pseudo-code in Figure 1. All outer loop iterations presenting an inner loop of size *f(i)* greater than lb_{THRES} are processed in a block-mapped fashion. If the block size is greater than *f(i)*, a large number of threads will not be assigned any work, leading to GPU underutilization. Therefore, in the remaining experiments we use small blocks consisting of 64 threads for the block-mapped kernels.

Results on SSSP: We now compare the performance of the five parallelization templates on *SSSP*. We refer to the basic implementation described in [5], which encodes the graph data structure in Compressed Sparse Row (*CSR*) format (more information can be found in [5]). When a graph is represented in *CSR* format, its traversal assumes the form of the nested loop in Figure 1(a), where the outer loop iterates over the nodes in the graph, and the inner loop over the neighbors of each node *i*. In irregular graphs where the node outdegree (*f[i]*) varies significantly from node to node, this traversal loop is irregular. Here, we show experiments performed on *CiteSeer*. Figure 5 shows the speedup of all code variants over the baseline thread-mapped implementation; the number of nested kernel calls performed by the two dynamic parallelism-based solutions are reported on top of the bars. Due to the irregular nature of the *CiteSeer* graph, almost all code variants that include load balancing outperform the basic thread-mapped implementation. The *dpar-naïve* code variant is the exception: due to the overhead of the large number of (small) nested kernel calls performed, this implementation leads consistently to (often significant) performance degradation. The delayed buffer-based and the optimized dynamic parallelism-based code variants yield the best results, and the performance improvement depends on the value of the load balancing threshold, which affects the amount of load balancing performed. The optimal value of this parameter corresponds to the warp size (no improvements were observed for lb_{THRES}<32).

To better understand these results, we used the Nvidia Visual Profiler to collect three performance metrics: the warp execution efficiency (ratio of the average active threads per warp to the maximum number of threads per warp on a streaming multiprocessor), and the global memory load and store efficiency (ratio of the number of requested memory load and store transactions to the actual number of load and store transactions performed - the lack of memory coalescing can cause more memory transactions than requested to be triggered). Table I shows the profiling data gathered for lb_{THRES} = 32. As can be seen, all parallelization templates but *dpar-naïve* report an increased warp efficiency compared to the baseline code. This indicates a better utilization of the available GPU cores. In addition, by processing nodes with low outdegrees (< lb_{THRES}) and nodes with high outdegrees separately, these code templates improve the memory load and store efficiency. Finally, thanks to its use of shared memory, *dbuf-shared* improves the memory coalescing over *dbuf-global*, leading to better memory efficiency. To conclude, our proposed parallelization templates improve both GPU core utilization and memory access patterns.

TABLE I. Profiling data collected on SSSP (lb_{THRES}=32)

Templates	Metrics		
	warp efficiency	*gld efficiency*	*gst efficiency*
baseline	35.6%	15.8%	3.2%
dual-queue	74.9%	79.1%	4.8%
dbuf-shared	75.7%	94.3%	50.4%
dbuf-global	72.3%	89.1%	8.5%
dpar-naïve	25.3%	45.5%	16.3%
dpar-opt	70.2%	63.2%	10.9%

Results on BC, PageRank and SpMV: Figure 6 shows the performance of *BC*, *PageRank* and *SpMV* using various lb_{THRES} settings. Again, we report the speedup achieved by our code variants over a thread-mapped implementation without load balancing. We make the following observations.

First, similarly to *SSSP*, the speedup decreases as lb_{THRES} increases. This behavior is not surprising: the load balancing threshold determines the number of iterations of the outer loop that are processed in a block-mapped fashion, thus reducing the warp divergence during the thread-mapped phase and the resulting core underutilization. In other words, the lower the value of lb_{THRES}, the more load balancing will take place through block-based mapping. Table II, which shows how the warp execution efficiency of *dbuf-shared* varies with lb_{THRES}, supports this observation. As can be seen, the lower the value of lb_{THRES}, the higher the warp efficiency (and, consequently, the GPU utilization). Note that the use of this parallelization template always improves the warp efficiency over the baseline code. We observed similar trends with all other parallelization templates but *dpar-naïve*.

Second, *dual-queue* performs better than the other code variants only on BC. *Dual-queue* suffers from the overhead of the initial creation of the two queues. While this overhead is limited for small datasets (e.g. *Wiki-vote* used by BC), its negative effect on performance becomes more obvious on large datasets (e.g. *CiteSeer* used by PageRank and SpMV).

Third, on these applications *dbuf-shared* performs worse than *dbuf-global* for low values of lb_{THRES} and reports better or comparable performance for $lb_{THRES} \geq 128$. This trend can be explained as follows. Recall that both parallelization templates use a delayed buffer to identify the large iterations of the outer loop that must be processed in the second, block-mapped phase. Since *dbuf-global* stores this buffer in global memory, the load gets redistributed across the thread-blocks during the second phase of the code (that is, the content of

TABLE II. Warp execution efficiency (*dbuf-shared*)

Applications	lb_{THRES}				
	32	**64**	**256**	**1024**	**baseline**
SSSP	75.6%	71.9%	45.3%	37.2%	35.6%
BC	75.8%	56.7%	17.1%	10.8%	10.3%
PageRank	91.5%	87.0%	63.4%	50.9%	50.8%
SpMV	94.4%	82.3%	71.5%	51.5%	51.0%

the delayed buffer is partitioned fairly across thread-blocks). However, this load redistribution across thread-blocks does not take place in the case of *dbuf-shared*, which stores the buffer in shared memory and performs a single kernel call. The probability of load imbalance across thread-blocks is higher for low values of lb_{THRES}, since in this case more work is added to the delayed buffer. However, when lb_{THRES} increases, less work is added to the delayed buffer, thus decreasing the probability of work imbalance across thread-blocks in the second phase of the code. When the amount of load balancing is low, *dbuf-global* suffers from the overhead of launching the second kernel. We used profiling data to support this observation. Specifically, we analyzed the warp occupancy of these code variants (that is, the ratio of the average active warps per active cycle to the maximum number of warps supported on a multiprocessor). For small values of lb_{THRES}, *dbuf-global* reports a higher warp occupancy compared to *dbuf-shared* (for example, for lb_{THRES}=32, the warp occupancy of *dbuf-shared* and *dbuf-global* is 18.3% and 26.9%, respectively), indicating a better hardware utilization.

Finally, we observe that *dpar-opt* performs similarly to (or slightly worse than) *dbuf-shared*. The nested kernel handling and launch overhead outshadows the benefit of dynamically remapping the work spawned within nested kernels to the GPU hardware.

To summarize, we make the following observations. First, the load balancing threshold lb_{THRES} is the dominant factor affecting the performance. Second, due to its queue consruction overhead, on large datasets the *dual-queue* template tend to by outperformed by the delayed-buffer-based and the *dpar-opt* solutions. Third, *dbuf-global* avoids the intra-block imbalance that penalizes *dbuf-shared* at the price of an additional kernel invocation to redistribute the workload among blocks. The *dbuf-shared* template performs similarly to *dpar-opt* without requiring hardware support for dynamic parallelism, and usually it achieves the best performance. Finally, dynamic parallelism must be used judiciously: the naïve use of this feature can lead to a high number of small grid invocations and suffer from a significant overhead, thus killing the performance.

C. Experiments on Recursive Algorithms

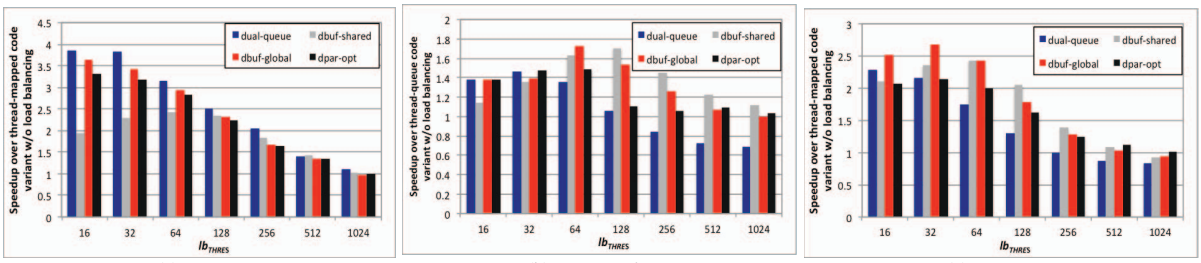
In section II.C we have discussed three GPU parallelization templates for recursive applications: non-recursive flat-parallelism, recursive-naïve and recursive-hierarchical (Figure 3). In this section we evaluate these parallelization templates on three applications that operate on tree and graph data structures: *Tree Height*, *Tree Descendants* and *recursive BFS*. In all the related charts we report the speedup of the code variants based on these parallelization templates over the better one between recursive and iterative serial CPU code.

Results on Tree Descendants/Heights: We evaluate the performance of Tree Descendants and Tree Heights on synthetic datasets. Our tree generator produces trees with different shapes based on three parameters: *tree depth*, *node outdegree* and sparsity. The sparsity parameter operates as follows. All non-leaf nodes have the same number of children, which is given by the node outdegree parameter. The probability ρ of the non-leaf nodes having children is defined as $\rho = (\frac{1}{2})^{\text{sparsity}}$. Therefore, if sparsity=0 all non-leaf nodes have children, leading to a regular tree where all leaf nodes have maximum depth; if sparsity=1 non-leaf nodes only possess a probability of ½ to have children, and so on. In general, larger values of the sparsity parameter lead to the generation of more irregular trees.

For both algorithms, we perform two sets of experiments over trees with different depths. First, we set the sparsity to 0 (regular trees) and vary the node outdegree from 32 to 512. This leads to trees that are regular in shape and exhibit an increasing level of parallelism at each node. Second, we set the node outdegree to 512 and vary the sparsity parameter from 0 to 4, thus allowing the generation of increasingly irregular trees. The results show that the depth parameter has no significant effect on performance because it does not change the irregularity and sparsity of the trees.

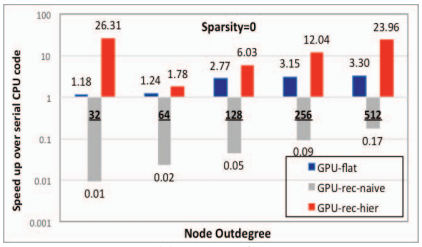
Figures 7(a) and (b) show the results reported by the *Tree Descendants* algorithm discussed in Section II.C (pseudo-code in Figure 3) on regular and irregular trees, respectively. Figure 7(c) shows the following profiling information for each code variant: warp utilization, number of dynamic kernel calls and number *dbuf-global* of atomic operations.

As can be seen in Figures 7(a) and (c), on regular trees the recursive-naïve implementation leads to significant performance degradation over the serial CPU code (and the

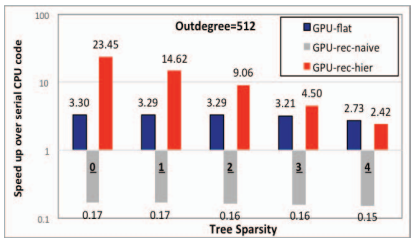


(a) *BC* (b) *PageRank* (c) *SpMV*

Figure 6. Speedup of the load balancing code variants over a basic thread-mapped implementation using various lb_{THRES} settings. For *BC*, the dataset is the *wiki-vote*. *PageRank* and *SpMV* are run on the *CiteSeer* network dataset. The naïve dynamic parallelism-based implementation (not shown for readability) is significantly slower than the other code variants.



(a) Sparsity = 0

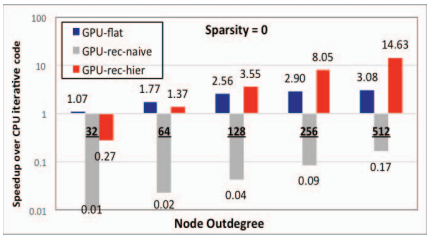


(b) Node outdegree = 512

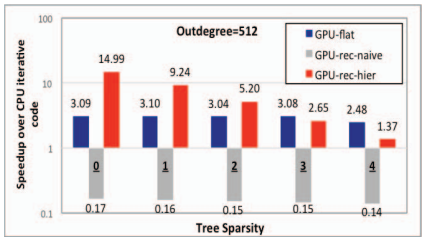
Out-degree	Flat-Kernel		Rec-naïve		Rec-hier		
	Warp	*Atomic*	*Warp*	*KCalls*	*Warp*	*Atomic*	*KCalls*
32	92.3%	0.10 m	32.0%	1.0k	68.0%	0.001m	33
64	93.5%	0.79 m	40.8%	4.1k	64.6%	0.004m	63
128	94.3%	6.32 m	59.7%	16k	65.2%	0.016m	129
256	94.4%	50.4 m	67.6%	66k	72.2%	0.065m	257
512	94.4%	403 m	74.4%	263k	84.4%	0.262m	513
Sparsity							
0	94.4%	403 m	74.4%	263k	84.4%	0.262m	513
1	94.4%	98.6 m	73.0%	64k	79.1%	0.129m	250
2	94.4%	26.0 m	70.4%	17k	71.9%	0.068m	132
3	94.3%	6.70 m	67.5%	4.4k	69.7%	0.035m	65
4	94%	1.68 m	66.0%	1.1k	69.1%	0.018m	35

(c) Profiling data

Figure 7. Tree descendants on synthetic trees with depth 4. Speedup of GPU code variants over iterative serial CPU code when (a) sparsity = 0 and (b) node outdegree = 512; (c) corresponding profiling information. The y-axis of charts (a) and (b) is in $\log_{10}$ scale: the values on top of the bars indicate the exact speedup numbers.



(a) Sparsity = 0



(b) Node outdegree = 512

Out-degree	Flat-Kernel		Rec-naïve		Rec-hier		
	Warp	*Atomic*	*Warp*	*KCalls*	*Warp*	*Atomic*	*KCalls*
32	94.2%	0.1m	32%	1.0k	67.1%	0.001m	33
64	95.6%	0.8m	40.9%	4.1k	68.6%	0.004m	63
128	96.2%	6.3m	59.8%	16k	70.3%	0.017m	129
256	96.3%	50m	67.8%	66k	76.1%	0.065m	257
512	96.3%	403m	74.6%	263k	86.2%	0.263m	513
Sparsity							
0	96.3%	403m	74.6%	263k	86.2%	0.263m	513
1	96.3%	98m	72.8%	64k	81.6%	0.129m	250
2	96.3%	26m	69.8%	17k	77.1%	0.068m	132
3	96.2%	6m	66.4%	4.4k	75%	0.035m	65
4	95.9%	1.6m	66.2%	1.1k	73.9%	0.018m	35

(c) Profiling data

Figure 8. Tree heights on synthetic trees with depth 4. Speedup of GPU code variants over iterative serial CPU code when (a) sparsity = 0 and (b) node outdegree = 512; (c) corresponding profiling information. The y-axis of charts (a) and (b) is in $\log_{10}$ scale: the values on top of the bars indicate the exact speedup numbers.

other GPU code variants) across all datasets. This is due to the large number of "small" nested kernel invocations as well as the low warp utilization of this code variant. The flat-parallelism template reports a decent speedup compared to the serial code when enough parallel work is spawned; however, due to the required atomic operations, its performance saturates for node outdegrees > 64. Despite the lower warp utilization, the recursive-hierarchical code variant outperforms the flat-parallelism template. This is due to the significant reduction in the number of atomic operations compared to the flat code.

As can be seen in Figures 7(b) and (c), the sparsity parameter does not significantly affect the performance and warp utilization of the flat-parallelism code variant. On the other hand, the performance of the recursive-hierarchical code decreases as the sparsity increases. As can be seen in Figure 7(c), in this case the more irregular structure of the tree has a negative effect on the warp utilization, causing this performance degradation.

Figure 8 shows the performance and profiling results reported by the *Tree Heights* algorithm on the same synthetically generated trees. We define the tree height in a recursive fashion as follows. Each leaf node within the tree is assigned height 1, and the height of a non-leaf node is defined as 1 + the maximum height across its children.

The recursive-naïve code variant achieves again poor performance across all datasets due to its low warp utilization and large number of small nested kernel launches. As can be seen in Figures 8(a) and (c), on regular trees the performance of the recursive-hierarchical kernel increases with the node outdegree. In fact, the node outdegree affects the amount of parallel work per node, and, as a consequence, the GPU utilization. This is confirmed by the warp utilization data. In addition, the recursive-hierarchical code variant achieves better performance than the flat-parallelism one for large node outdegrees. This is due to its significantly reduced number of atomic operations. As in Tree Descendants, the atomic operations cause the saturation of the performance of flat-parallelism beyond a node outdegree of 128.

As shown in Figures 8(b) and (c), when the sparsity increases, the behavior of Tree Heights is very similar to that of Tree Descendants. Specifically, as the tree becomes more irregular, the warp divergence causes the warp utilization of the hierarchical kernel to drop significantly; on the other hand, when the tree gets sparser, the atomic operations required by the flat-parallelism kernel are reduced and its

speedup stays stable, making the flat kernel preferable to the recursive hierarchical one.

Results on recursive BFS: We now want to evaluate our parallelization templates on a graph algorithm and compare the results with those obtained on the two tree traversal applications described above. To this end, we apply our parallelization templates to BFS. We perform experiments on randomly generated graphs consisting of 50,000 nodes. In this case, the node outdegree is uniformly distributed within a variable range (x-axis of Figure 9). This leads to a number of edges varying from 1.6 to about 27 million, while the number of levels in the graphs varies from 4 to 5. Our flat GPU implementation is a thread-mapped parallelization of level-based BFS traversal [5]. Our recursive implementations are unordered [11]: the traversal of a node causes the recursive traversal of its neighbors as long as their level decreases. This implementation is not work-efficient and, due to stack serialization, causes serial CPU traversal to happen in depth-first (rather than in breadth-first) fashion. Nevertheless, on CPU the recursive implementation outperforms the iterative one by a factor varying from 1.25x to 3.3x depending on the graph size. On GPU, the *flat* code variant outperforms the CPU recursive implementation by a factor 11-14x, whereas both recursive implementations lead to a considerable slowdown (in the order of 700-14,000x). This behavior significantly deviates from what observed on tree traversal algorithms and can be explained as follows. First, in this case the flat parallel code variant does not require atomic operations, while the recursive code variants do. Second, the considered tree algorithms are work efficient: they traverse each node exactly once. Since in a graph multiple nodes can share neighbors, this property does not hold for unordered BFS traversals.

In Figure 9, we compare the results obtained applying the two recursive GPU parallelization templates. In each case, we also test using multiple streams per thread-blocks, thus allowing concurrent execution of nested kernel calls originated from the same block. We observe performance improvements only when one additional stream per thread-block is created (more streams cause overhead without leading to significant parallelization benefits). In Figure 9, we use the suffix "*stream*" to refer to code variants using an additional stream per block. We make the following observations. When only the default stream is used, the hierarchical is again preferable to the naïve implementation. On the other hand, while multiple stream support helps the naïve implementation, it is detrimental for the hierarchical code variant. These results can be explained as follows. In the naïve case, the parallel execution of small nested kernels allows better GPU utilization and reduction in the number of kernel calls (by anticipating the processing of nodes closer to the root in a BFS fashion, the traversal becomes more work efficient). The hierarchical implementation, however, allows parallel execution of kernels even in the absence of additional streams, since kernels spawned by different thread-blocks can be executed concurrently. In this situation, the stream handling overhead is not overshadowed by a better GPU utilization, and the parallel execution of many kernels can make the traversal less work-efficient, especially in case of large node outdegrees.

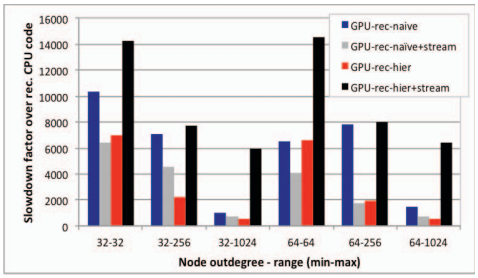


Figure 9. Recursive BFS computation: slowdown of the GPU code variants over recursive serial CPU code. The experiments are performed on randomly generated graphs consisting of 50,000 nodes. The node outdegree is randomly generated within the indicated range.

Finally, we have tested the use of multiple streams on tree traversal. This optimization increases the performance of the naïve recursive parallelization template. However, the performance improvement is in this case more moderate than in graph traversal. Multiple streams allow parallel execution of kernel launches and better GPU utilization. However, they do not make the tree traversal more work efficient (every node is processed only once independent of the number of CUDA streams used). The use of multiple streams does not have a significant effect on the hierarchical recursive parallelization template, which has a good GPU utilization even with a single stream and remains the preferred solution.

IV. Related Work

As GPUs have become more general purpose, the interest of the research community has moved toward effectively deploying irregular applications on these many-core platforms. In particular, there have been several efforts focusing on the acceleration of graph processing algorithms on Nvidia GPUs [3, 5, 12-24]. Harish and Narayanan [5] proposed a basic implementation suited to regular, dense graphs. On sparse graphs, better results have been reported in subsequent efforts, which covered a variety of algorithms (breadth-first search [12-18], single-source shortest path [16-18, 20], minimum spanning tree [17, 19], Delaunay mesh refinement [16-19], points-to analysis [16, 18, 19, 21], strongly connected components [22] and survey propagation [16-19]). Among these, Hong et al. [20] proposed a virtual warp-centric programming model with more general applicability; Burtscher et al. [16-19] and Che et al. [23] proposed benchmark suites of graph algorithms and identified computational patterns common to different graph applications. In this work we target irregular applications beyond graph algorithms. In addition, rather than fine tuning specific applications, we aim to provide and evaluate parallelization templates of general applicability.

A few research efforts have addressed the problem of nested parallelism on SIMD architectures. Blelloch and Sabot [25] have proposed *flattening* transformation techniques aimed to vectorize programs with nested data parallelism, so to allow their deployment onto hardware designed to accommodate a single level of parallelism (e.g., SIMD processors). They validated their compilation

techniques on their proposed NESL language [26]. More recently, Bergstrom and Reppy [27] ported NESL to GPUs. In particular, they relied on the NESL compiler for the flattening transformation and explored the design of data structures and primitives (e.g., element-wise instructions, scans, reductions, permutations) that enable good performance on GPU. Their proposed flattening techniques can be used to deploy recursive applications on GPUs without support for nested kernel invocations (in other words, to produce flat parallelization templates).

Recent studies have analyzed the strengths and limitations of Nvidia's dynamic parallelism. Yang and Zhou [1] have proposed compiler techniques to implement nested-thread level parallelism on algorithms operating on small datasets where dynamic parallelism would perform poorly. Their solution, however, would be less effective on large datasets, which we consider in this work. The effectiveness of dynamic parallelism on larger datasets has been demonstrated on different applications: clustering algorithms [28], computation of the Mandelbrot set [29] and a particle physics simulation [30]. A recent study [2] has proposed a characterization of dynamic parallelism on unstructured applications. In particular, the proposed characterization focus on modeling the effect of control and memory access patterns on the performance. Differently from this work, the considered applications are mostly parallelized as in our naïve recursive parallelization template.

V. CONCLUSION

In this paper we have studied two computational patterns exhibiting nested parallelism: irregular nested loops and parallel recursive computations. We have proposed several parallelization templates to better distribute the work to the GPU hardware resources, and we have evaluated the use of nested parallelism on the considered computational patterns. Our experiments show that, by carefully selecting the parallelization template, applications with irregular nested loops can achieve a 2-6x speedup over basic thread-mapped GPU implementations. The use of nested parallelism on recursive tree traversal can lead to significant speedup (up to a 15-24x factor) over iterative serial CPU code, and can be preferable to the parallelization of iterative versions of these algorithms. However, the benefits of nested parallelism on recursive applications operating on array and graph data structures are still unclear, especially when recursive code variants require synchronization through atomic operations.

VI. ACKNOWLEDGMENTS

This work has been supported by NSF awards CNS-1216756 and CCF-1452454 and by a gift from NEC Laboratories America and equipment donations from Nvidia Corporation.

REFERENCES

[1] Y. Yang, and H. Zhou, "CUDA-NP: realizing nested thread-level parallelism in GPGPU applications," in Proc. of PPoPP 2014.

[2] J. Wang, and S. Yalamanchili, "Characterization and Analysis of Dynamic Parallelism in Unstructured GPU Applications," in Proc. of IISWC 2014.

[3] D. Li, and M. Becchi, "Deploying Graph Algorithms on GPUs: an Adaptive Solution," in Proc. of IPDPS 2013.

[4] M. Goldfarb, Y. Jo, and M. Kulkarni, "General transformations for GPU execution of tree traversals," in Proc. of SC 2013.

[5] P. Harish, and P. J. Narayanan, "Accelerating large graph algorithms on the GPU using CUDA," in Proc. of HiPC 2007.

[6] A. E. Sriyuce, et al. , "Betweenness Centrality on GPUs and Heterogeneous Architectures." in Proc. of GPGPU-6 2013.

[7] N. T. Duong, et al., "Parallel PageRank computation using GPUs." in Proc of 3rd Symp. on Information and Communication Technology, 2012.

[8] J. L. Greathouse, and M. Daga, "Efficient sparse matrix-vector multiplication on GPUs using the CSR storage format." in Proc of SC 2014.

[9] "DIMACS Implementation Challenges," http://dimacs.rutgers.edu/Challenges/.

[10] "Stanford Large Network Dataset Collection," http://snap.stanford.edu/data.

[11] M. A. Hassaan, M. Burtscher, and K. Pingali, "Ordered vs. unordered: a comparison of parallelism and work-efficiency in irregular algorithms," in Proc. of PPoPP 2011.

[12] L. Luo, M. Wong, and W.-m. Hwu, "An effective GPU implementation of breadth-first search," in Proc. of DAC 2010.

[13] D. Merrill, M. Garland, and A. Grimshaw, "Scalable GPU graph traversal," in Proc. of PPoPP 2012.

[14] S. Hong, T. Oguntebi, and K. Olukotun, "Efficient Parallel Graph Exploration on Multi-Core CPU and GPU," in Proc. of PACT 2011.

[15] A. Gharaibeh, et al., "On Graphs, GPUs, and Blind Dating: A Workload to Processor Matchmaking Quest," in Proc. of IPDPS 2013.

[16] M. Burtscher, R. Nasre, and K. Pingali, "A quantitative study of irregular programs on GPUs," in Proc. of IISWC 2012.

[17] R. Nasre, M. Burtscher, and K. Pingali, "Data-driven versus Topology-driven Irregular Computations on GPUs," in Proc. of IPDPS 2013.

[18] R. Nasre, M. Burtscher, and K. Pingali, "Atomic-free irregular computations on GPUs," in Proc. of GPGPU 2013.

[19] R. Nasre, M. Burtscher, and K. Pingali, "Morph algorithms on GPUs," in Proc. of PPoPP 2013.

[20] S. Hong, et al., "Accelerating CUDA graph algorithms at maximum warp," in Proc. of PPoPP 2011.

[21] M. Mendez-Lojo, M. Burtscher, and K. Pingali, "A GPU implementation of inclusion-based points-to analysis," in Proc. of PPoPP 2012.

[22] J. Barnat, et al., "Computing Strongly Connected Components in Parallel on CUDA," in Proc. of IPDPS 2011.

[23] C. Shuai, et al., "Pannotia: Understanding irregular GPGPU graph applications," in Proc of IISWC 2013.

[24] F. Khorasani, et al., "CuSha: vertex-centric graph processing on GPUs," in Proc. of HPDC 2014.

[25] G. E. Blelloch, and G. W. Sabot, "Compiling collection-oriented languages onto massively parallel computers," J. Parallel Distrib. Comput., vol. 8, no. 2, pp. 119-134, 1990.

[26] G. E. Blelloch, NESL: A Nested Data-Parallel Language, Carnegie Mellon University, 1992.

[27] L. Bergstrom, and J. Reppy, "Nested data-parallelism on the gpu," in Proc. of ICFP 2012.

[28] J. DiMarco, and M. Taufer, "Performance Impact of Dynamic Parallelism on Different Clustering Algorithms and the New GPU Architecture," in Proc. of SPIE Defense, Security, and Sensing Symp. 2013.

[29] A. Adinetz. "Adaptive Parallel Computation with CUDA Dynamic Parallelism," http://devblogs.nvidia.com/parallelforall/introduction-cuda-dynamic-parallelism/.

[30] A. Adinetz. "A CUDA Dynamic Parallelism Case Study: PANDA," http://devblogs.nvidia.com/parallelforall/a-cuda-dynamic-parallelism-case-study-panda/.