Compiler-Assisted Workload Consolidation to Efficiently Exploit Dynamic Parallelism for Recursive Applications

---

A Thesis

presented to

the Faculty of the Graduate School

at the University of Missouri-Columbia

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

---

by

HANCHENG WU

Dr. Michela Becchi, Thesis Supervisor

DECEMBER 2015

The undersigned, appointed by the dean of the Graduate School, have examined the thesis entitled

Compiler-Assisted Workload Consolidation to Efficiently Exploit Dynamic Parallelism for Recursive Applications

presented by Hancheng Wu,

a candidate for the degree of master of science,

and hereby certify that, in their opinion, it is worthy of acceptance.

Professor Michela Becchi

Professor Guilherme DeSouza

Professor William Harrison

ACKNOWLEDGEMENTS

I would like to express my very great appreciation to Dr. Michela Becchi for her encouragement, guidance and support throughout my time in the lab.

I would like to give my thanks to Dr. DeSouza and Dr. Harrison for their valuable time at my thesis defense.

I want to thank all the lab members who always help me and support me.

I would like to offer my special thanks to Mengxuan Ma, who just makes my life beautiful.

Last, I want to thank my parents for everything.

TABLE OF CONTENTS

List of Figures

Abstract

GPUs have been widely used to parallelize and accelerate applications for its high throughput. Traditionally, a GPU function can only be launched from the CPU side. This results in the fact that GPUs are preferable for those application which express a flat data parallelism, a simple data parallelism that is known at compiling time and can be easily distributed to different GPU blocks and threads. However, for those applications that contain nested data parallelism, which is not known *a priori* and can only be discovered at running time, it is difficult to write a GPU function that achieve high performance on parallelization and acceleration. One can easily end up with either a too coarse-grained or too fine-grained GPU function.

Since Kepler architecture, Nvidia introduced a new feature--Dynamic Parallelism (DP), which enables the initiation of GPU functions from inside a GPU function. This makes the nested parallelism easy to be explored on GPU since one can program in a way that a new GPU function can be launched whenever a local nested parallelism is met during the execution. What is more, DP makes implementing recursion on GPU without the intervention of CPUs possible.

Many computations exhibit a pattern of nested data parallelism and among those is parallel recursion. However, preliminary data shows that simple DP-based implementations of recursion result in poor performance. This work focus on how to efficiently exploit DP for parallel recursive applications on GPU. Specifically, the goal is to free the users from programming with the complexity of GPUs' hardware and software and to automatically generate high performance GPU recursive functions implemented

with DP given the inputs of simple parallel CPU recursive functions. To this end, first, I propose several DP-based parallel recursive templates that can be generated from a serial CPU recursive function. I compare the parallel recursive templates with non DP-based counterparts (flat kernels) to see if using DP in parallel recursive application can be beneficial or not. Second, to reduce the overhead of DP, I propose compiler techniques that improve the efficiency of simple DP-based parallel recursive functions by performing workload consolidation.

My evaluation shows that GPU kernels consolidated with the proposed code transformations achieve an average speedup in the order of 1500x over basic implementations using DP and an average speedup of 3.9x over optimized flat GPU kernels for both tree traversal and graph based applications.

Chapter 1 Introduction

Graphics Processing Units (GPUs) have proven to successfully accelerate regularly structured, data intensive applications. Applications that naturally map onto the GPU hardware typically exhibit simple parallelism patterns – such as flat and two-level parallelism – and a degree of parallelism that can be determined prior to kernel launch based on the size of the inputs. However, many common applications operate on irregularly structured data (such as graphs and meshes) and present data-dependent control flows and irregular memory access patterns, resulting in unpredictable runtime behaviors. Examples of such applications include recursive algorithms, adaptive meshes and computations with irregular loops [1]. The use of flat parallelism to implement these applications can lead to uneven work distribution across threads and, ultimately, to inefficient codes. In fact, in the presence of flat kernels, fine-grained partitioning of the work across threads may lead to a large number of idle or lightly utilized threads, while coarse-grained partitioning of the work may cause uneven work distribution among threads. On the other hand, the use of more adaptive parallelism patterns, such as nested parallelism, can better capture the dynamic nature of the workload within unstructured applications.

Since the Kepler architecture, Nvidia has introduced in its GPUs a new feature, called dynamic parallelism (DP), which makes it possible for GPU threads to dynamically spawn GPU kernels. Dynamic parallelism has also been recently added to the OpenCL 2.0 standard. By supporting nested kernel invocations, this feature enables and facilitates parallelization of recursive algorithms as well as dynamic load balancing and data-

dependent execution. However, the effective use of DP is not a trivial matter. Basic DP implementations of recursion that spawn recursive kernels on a per-thread basis whenever recursion is locally generated tend to perform a large number of small kernel launches. It has been shown that, due to the runtime overhead associated with nested kernel calls, these implementations often lead to significantly degraded performance, even worse than those of flat parallel variants of the same algorithms [2, 3].

In this work, I focus on the efficient use of DP for parallel recursive applications. First, to see if using DP in parallel recursive functions is beneficial, I propose several parallelization templates that can be automatically generated from a serial CPU recursive function by a compiler and then compare the DP-based parallel recursive templates with non DP-based counterparts. I find out that to make DP-based implementations beneficial, one should try to use few nested kernel invocations that spawn a significant amount of parallel work. However, most of the basic DP-based implementations will not satisfy this requirement. To reduce the overhead of DP, I then propose compiler techniques that can improve the efficiency of basic DP-based parallel recursive functions by performing workload consolidation, reducing the overhead of DP. Specifically, my proposed workload consolidation scheme consolidates the workload belonging to kernels that would be spawned by multiple GPU threads into a single nested kernel. I consider performing kernel consolidation at three granularities: warp-, block- and grid-level, whereby the consolidation involves the kernels launched by all the threads within a warp, all the threads within a block or the entire grid, respectively. I evaluate these consolidation mechanisms on three recursive applications.

Chapter 2 Background

2.1. GPU

Graphics Processing Units (GPUs) have widely used to accelerate applications from various domains [4-7]. Different than CPUs, the design of GPUs adopts a many-core architecture, where hundreds or even thousands of cores may be present. On the hardware side, a GPU consists of certain number of Streaming Multiprocessors (SMs) and each SM consists of many scalar processors. All SMs/cores have access to global memory that resides on the GPU chip, while each SM has its own fast memory that can be configured either as shared memory used by threads running on the SM or as L1 cache. On the software side, Nvidia provides a software stack called CUDA for users to program on GPU. Basically, CUDA abstracts GPU's hardware into blocks and threads inside each block. A block is a virtualized SM and a thread is a virtualized scalar processor. A block is then mapped to a SM for execution and threads inside the block are mapped to the scalar processors on that SM. A SM may hold many blocks at a time as long as resources (such as registers for thread and shared memory for blocks) on the SM are sufficient to run that many blocks. A GPU function, which is also called a GPU kernel, is initiated with a configuration that specifies the number of blocks and the number of threads per block for the execution. Threads seem to execute the instructions in a Single Instruction Multiple Devices (SIMD) manner. From the scheduling perspective, threads running on a SM are scheduled based on groups of 32 (warps), however no order of the execution of the warps is guaranteed, as a result, the correctness of the kernel should not be depending on how these warps are scheduled. Threads inside a warp is automatically synchronized

because of the SIMD execution manner and threads inside a block can be synchronized through CUDA library calls, however there is no library function call for a global synchronization. In a case where global synchronization cannot be avoided, programmers have to implement a customized global synchronization method which will possibly introduce a deadlock if not well taken care of.

2.2. Dynamic Parallelism

Traditionally, only CPU threads can launch GPU kernels. Dynamic Parallelism, a feature added to OpenCL 2.0 standard and supported by Nvidia GPUs with compute capability 3.5 and above, makes it possible for GPU threads to launch GPU kernels. Kernel launches can be nested and the deepest nesting level supported is currently 24. Kernels launched from different blocks or streams can execute concurrently, and up to 32 concurrent kernels are currently allowed on Nvidia GPUs. A parent kernel will return only after all its child kernels have completed; however, the order of their execution is unknown unless these kernels are explicitly synchronized by a *cudaDeviceSynchronize* call. For each nesting level up to an explicit synchronization, parent kernels may be temporarily swapped out to free up GPU resources and allow the execution of their child kernels. Pending kernels due to either unresolved dependencies or lack of available hardware resources are fed to a temporary buffer. Global memory data are visible to both parent and child kernels, while shared and local memory variables are visible only within the kernel where they are declared.

2.3. Parallel Recursion

Parallelization of recursive algorithms on GPU is an interesting topic. While some recursive algorithms can be made iterative through various code transformation

techniques (e.g. tail-recursion elimination, auto-ropes and other flattening techniques [8-10]) and subsequently undergo flat parallelization, recursion cannot be always eliminated. Before the introduction of DP on GPU, parallel recursion required both CPU and GPU intervention. Specifically, the CPU would control the flow of recursion and initiate the recursive calls, whose execution could then be offloaded to the GPU in the form of parallel kernels. This approach requires one kernel launch for every recursive call and incurs high CPU-GPU communication overhead. By enabling kernel launches from GPU threads, DP allows the recursive control flow to reside directly on the GPU. This, in turn, allows reducing the CPU-GPU communication and the kernel launch overhead. The most natural way to implement a parallel recursive function on GPU is by directly porting to this platform a CPU parallelization of that function and allowing each GPU thread to spawn a recursive kernel whenever the CPU code would perform a recursive call. As will be shown, GPU implementations following this pattern often perform a large number of small kernel invocations, leading to substantial kernel launch overhead and hardware underutilization. The proposed consolidation schemes in later chapters target this problem.

Chapter 3 Motivation

In this section, I first present the basic use case of dynamic parallelism for parallel recursion. Then I show the basic implementation may lead to poor performance and explain the overhead of using DP. This motivates me to study in which situation using DP-based implementation for recursion is preferable and study how to improve the performance of DP-based implementations.

3.1. Basic DP-Code Template

Figure 1(a) shows a basic code template for parallel recursive kernels that use DP [2]. As in all GPU kernels, each thread (or thread-block) is assigned some data to process (or work items). Each thread (thread-block) initially performs some work (*prework*) on the data assigned to it. Then, depending on the outcome of a condition, the thread (thread-block) either spawns a child kernel to execute some more work, or performs that work on its own. Finally, the thread (thread-block) may optionally perform additional work (*postwork*). Figures 1(b) illustrate this basic code template on recursive tree traversal.

In the tree traversal kernel, each thread is assigned a child of a given node. Initially, the thread checks whether the child has children or not. In the case of no children, the thread performs *leafnode_work*; otherwise, it spawns a kernel recursively and delegates to it the processing of its assigned node.

The example above illustrates a “basic” implementation of parallel recursion that relies on DP. This basic code variant results from simply porting the CPU implementation of a parallel recursive function to GPU. Although more complex

```
__global__ void rec_kernel() {
   work_item= get_work_item(…)
   prework(work_item)
   if (condition)
      rec_kernel<<<block_dim, thread_dim>>>(…, work_item, …)
   else
      work(work_item)
   postwork(work_item)
}
```

(a) Basic DP template for recursive kernels

```
__global__ void tree_traversal( node_t node) {
   int i = threadIdx.x
   node_t child = node.children[i]
   if (child.children.size>0)
      tree_traversal<<<1, child.children.size>>>(child)
   else {
      leafnode_work()
   }
   postwork()
}
```

(b) Basic implementation of tree traversal with DP

Figure 1 Basic-dp code template and sample code

implementations are possible, these basic code variants require minimal effort from the programmer.

3.2. Limitations of Dynamic Parallelism

The effectiveness of DP is affected by different factors: sources of runtime overhead and GPU utilization. Below, I detail each of these aspects.

Kernel Launch Overhead - To launch a kernel, the CUDA driver and runtime need to parse the parameters list, buffer the values of these parameters, and dispatch the kernel. These steps have an associated overhead. This overhead is negligible when the number of nested kernels is small, but can accumulate and become significant in the presence of numerous kernel launches [2, 3].

Kernel Buffering Overhead - Kernels waiting to execute are inserted in a pending buffer. Since CUDA 6, this buffer consists of two pools: a fixed-size pool and a variable-size virtualized pool. The fixed-size pool incurs lower management overhead but by default can only accommodate a maximum of 2048 pending kernels. When it becomes full, pending kernels are fed to the virtual pool, which incurs extra management overhead. Applications spawning a large number of nested kernels can exhaust the fixed-size pool and experience performance degradation due to the virtual pool's overhead. It is possible to increase the capacity of the fixed-size pool through the CUDA function *cudaDeviceSetLimit*. However, this will result in a higher global memory reservation.

Synchronization Overhead - If there exists explicit synchronization between parent and child kernels, in order to free resources for the execution of child kernels, parent kernels will be swapped out into global memory. For each level up to the maximum level where they synchronize, up to 150 MB memory may be reserved for swapping. These extra memory transactions are source of additional overhead.

Effect of the kernel configuration – It is well known that the full use of the GPU hardware requires massive multithreading. CUDA currently allows up to 32 kernels to execute concurrently on a GPU. However, if configured to use small thread configurations, even 32 concurrent kernels may underutilize the GPU. Meanwhile, there is a limit on the maximum number of blocks that can be concurrently active. Thus, configuring nested kernels with a large number of blocks will limit the number of kernels executing in parallel. As a result, it is important for programmers to carefully select thread configurations that allow both good device utilization and desired level of concurrency.

The kernel launch, buffering and synchronization overheads can be reduced by limiting the number of kernel launches performed. In addition, to avoid GPU underutilization, it is important to avoid small kernel configurations that would lead to low occupancy even in the presence of kernel concurrency.

In general, DP codes resulting in a large number of small kernel invocations tend to experience poor performance. Preliminary result shows that due to the large number of small kernel calls they invoke, DP codes for tree traversals following the basic template in Figure 1 can underperform flat implementations of the same algorithms by up to a factor of 1000.

Chapter 4 Methodology

My final goal is to automatically generate high performance DP-based GPU recursive functions given inputs of simple parallel CPU recursive functions, freeing the users from programming with the complexity of GPUs' architecture and handling of DP's overhead. To this end, my first step is to understand in which situations implementing recursive kernels using DP is preferable to non-recursive flat counterpart kernels of the algorithms considered. I propose several parallelization recursive templates that can be automatically generated from a serial CPU recursive function by a compiler. I compare the DP-based parallel recursive templates with non DP-based counterparts. I find that to make DP-based implementations beneficial, one should try to use few nested kernel invocations that spawn a significant amount of parallel work. However, most of the basic DP-based implementations will not satisfy this requirement. To reduce the overhead of DP, I propose compiler techniques that can improve the efficiency of basic DP-based parallel recursive functions by performing workload consolidation, reducing the overhead of DP.

In this chapter, I first introduce the different parallel recursive templates that I use to study if DP implementations can be beneficial; second I will introduce the proposed compiler techniques of workload consolidation that can improve the performance of basic DP-based implementation.

4.1. Parallelization templates

As illustration, I consider the recursive computation of the number of descendants in a tree. Figure 2(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 2(b) can be obtained by applying recursion elimination techniques [10]. The flat kernel in Figure 2(c) parallelizes the iterative version of the code using thread-

(a) recursive serial code

```
unsigned rec_function(graph g, node n){
  if (!leaf(n))
    for (node c ∈ g.children(n))
      g.descendants[n]+=rec_function(g,c);
  return g.descendants[n];
}
```

(b) iterative serial code

```
void 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

```
void 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 – basic DP approach

```
void 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

```
void 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_SM=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 2 Application of different parallelization templates for recursive algorithms to the computation of tree descendants.

based mapping. I consider two parallel recursive code variants. The basic DP template in Figure 2(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 2(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. Recursion is performed 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). This hierarchical recursive template can only be applied to tree and graph applications.

In Chapter 5.2, I evaluate the proposed parallelization templates on different recursive algorithms.

4.2. Workload Consolidation

After studying the templates in Chapter 4.1, I find out that to make DP-based implementations beneficial, one should try to use few nested kernel invocations that spawn a significant amount of parallel work (like the hierarchical recursive template for trees/graphs). However, most of the basic DP-based implementations as shown in Figure 1 and Figure 2(d) will not satisfy this requirement. In this section, I present compiler techniques of workload consolidation to avoid the performance degradation associated with the basic implementation of dynamic parallelism.

4.2.1. Workload Consolidation

The idea at the basis of workload consolidation is fairly simple: by aggregating kernels spawned by different threads into a single or few consolidated kernels, it is possible both to decrease the number of nested kernel invocations, thus limiting DP overhead, and to increase the degree of multithreading of the nested kernels invoked, thus increasing their GPU utilization. In order to perform workload consolidation, I buffer the work associated to the kernels to be consolidated, and I defer the handling of this aggregated work to the launch of one or more child kernels. Since in the SIMT model all threads execute the same instructions on different data, in order for a thread to buffer work, it will be sufficient for the thread to buffer the pointer(s) or index(es) to the data to be processed. This method requires barrier synchronization between the buffer insertions and the

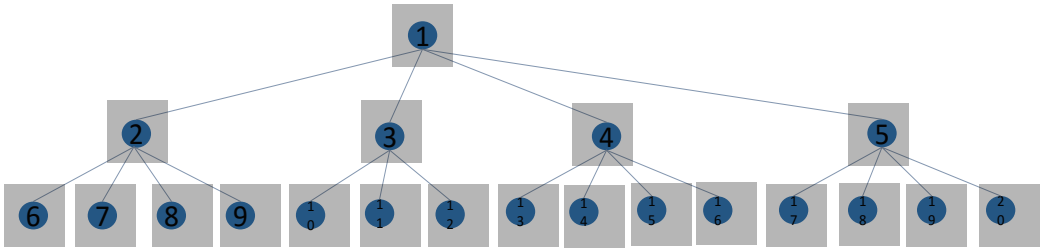


Figure 3(a). Recursive tree traversal with basic DP implementation

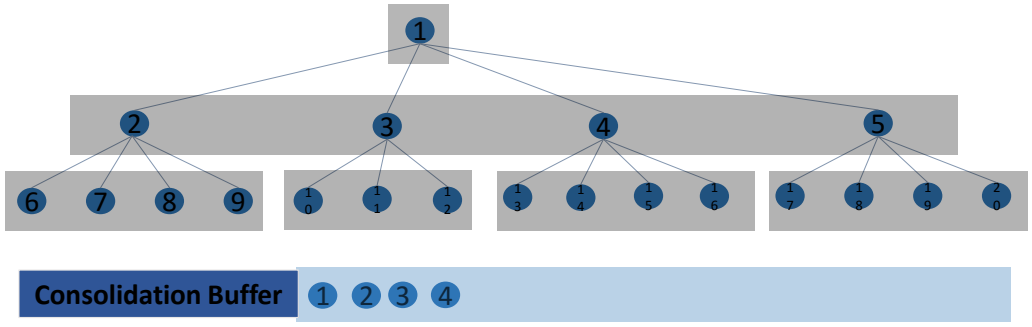


Figure 3(b). Dynamic parallelism with workload consolidation

Figure 3 Workload Consolidation

consolidated kernel launches.

This high level idea is illustrated with a tree traversal algorithm in Figure 3. In the figure, the dark circle indicate a tree node with certain amount of work assigned to it and each grey rectangular indicates a GPU kernel. Figure 3(a) shows the basic DP implementation of the tree traversal algorithm where each tree node is processed by a GPU kernel; when the kernel sees that the current node has children, it will spawn a child kernel for each of the children. As can be seen, the number of concurrent kernel invocations grows exponentially. The workload consolidation scheme illustrated in Figure 3(b), instead of launching a nested kernel for every child, first inserts the children of a node into a consolidation buffer, and then launches a single consolidated child kernel to process all the nodes in the buffer. The consolidated child kernel will have a larger thread/block configuration than the nested kernels in Figure 3(a). As can be seen, the number of child kernel invocations (number of grey rectangular) reduces from 20 in (a) to 6 in (b).

4.2.2. Consolidation Granularity

CUDA programming model has four levels of parallelism: thread, warp, block and grid. While threads, blocks and grids are exposed to programmers, warps are implicitly defined as groups of 32 threads that execute in lockstep and have contiguous identifiers. In the DP execution model, kernel launches are performed by threads. I consider three consolidation granularities: warp-, block- and grid-level.

Warp-level consolidation uses a buffer to aggregate work from the threads within a warp and launches one kernel per warp. This consolidation method can reduce the number of kernel invocations at most by a factor of 32. The benefit of warp-level

consolidation is that the synchronization overhead is minimized because no additional synchronization is required beside the implicit one due to SIMD execution.

Block-level consolidation aggregates work associated to threads within a block and launches a kernel per block. This method can reduce the number of kernel invocations beyond what allowed by warp-level consolidation. However, this consolidation scheme requires a block-level synchronization (*__syncthreads*) after the buffer insertions, leading to higher synchronization overhead than the warp-level variant.

Grid-level consolidation aggregates work from all threads in a grid and then launches a single child kernel. Since CUDA does not provide global synchronization within a kernel, this consolidation method requires a customized barrier synchronization (will be discussed later in detail). Because of this, grid-level consolidation suffers from the highest synchronization overhead.

4.2.3. Kernel Transformations

The overall kernel transformation flow is shown in Figure 4. The input is DP-based CUDA code annotated with the described pragma directive, and the output is the consolidated CUDA code. The kernel transformation process consists of two steps: (1) child step and (2) parent step transformation. The two transformation steps are applied to the single input recursive kernel sequentially.

Child step transformation – This phase transforms the input kernel with a focus on how to recover the processing of the workload from the consolidation buffer. The new consolidated kernel fetches work from the consolidation buffer and processes that work according to the code in the input kernel. Whenever possible, I generate moldable kernels [11] (that is, kernels with tunable kernel configuration). The way the original code is

mapped to threads and blocks in the consolidated kernel depends on the configuration of the input child kernel. Specifically, I identify the following cases:

Solo thread: The input kernel consists of a single block and a single thread (e.g. quick sort in CUDA SDK). In the consolidated child kernel, each thread will fetch a work item

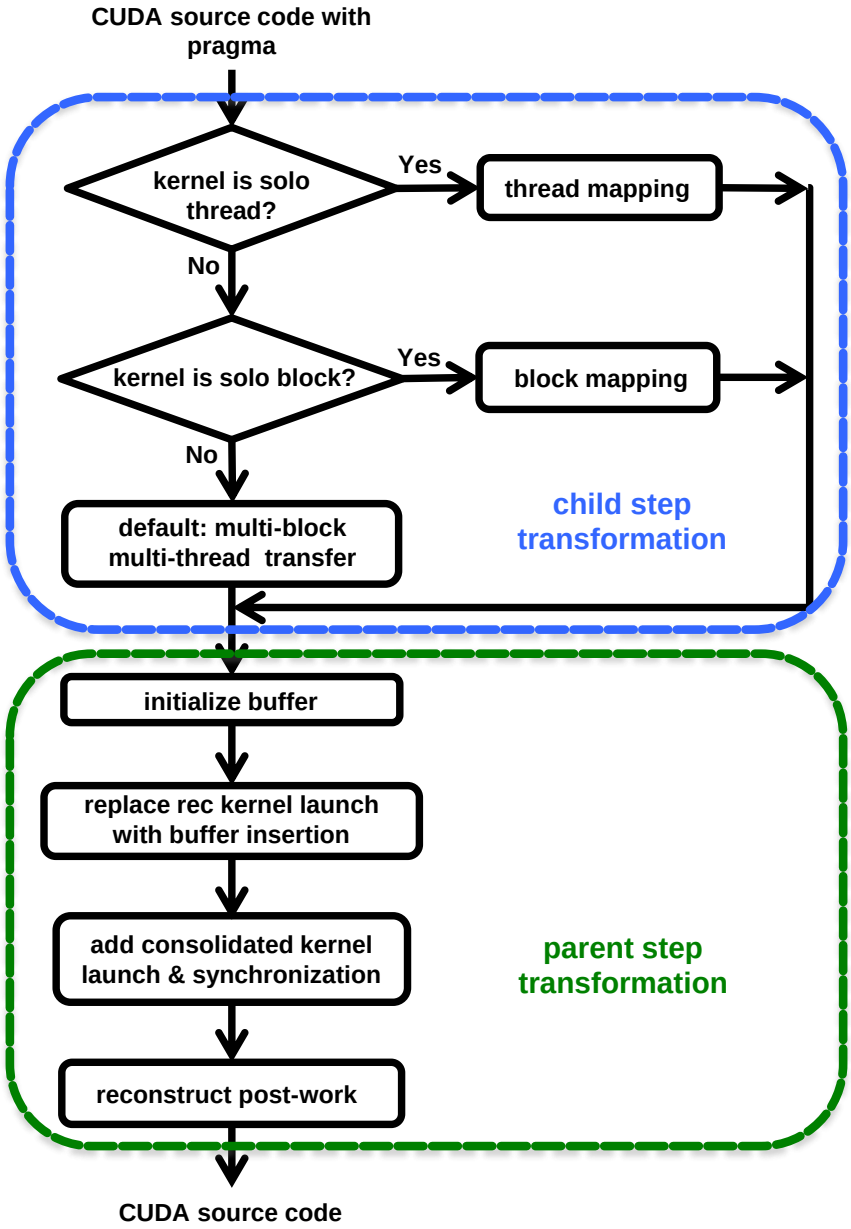


Figure 4 Kernel Transformation flow

(if available) from the consolidation buffer, and it will process that work item exactly as the original kernel does. To make the kernel moldable, I allow threads to fetch work from the buffer repeatedly until the buffer becomes empty.

Solo block: The input kernel consists of a single block with multiple threads, and these threads operate cooperatively. In the consolidated child kernel, each block will fetch a work item (if available) from the consolidation buffer, and the threads within the block will cooperatively process that work item as in the original kernel. To make the kernel moldable, I allow blocks to fetch work from the buffer repeatedly until the buffer becomes empty. If the original child kernel is moldable, the number of threads per block in the generated child kernel will also be tunable; else, the two kernels will have the same block size.

Multi-block and multi-thread: When the original kernel uses multiple blocks and threads per block, each work item is processed by all threads cooperatively. In this case, in the transformed kernel I use a for-loop to wrap the code of the original child kernel. The generated kernel will extract work from the buffer iteratively, and all threads will process cooperatively each work item. In this case, the transformed kernel is moldable only if the original kernel is such.

Parent step transformation – The input kernel is divided into three sections: *prework*, *recursive kernel launch*, and *postwork* (Figure 1(a)). The prework and postwork represent the processing done before and after the recursive kernel launch, respectively. Many kernels do not include any postwork (BFS). The code transformations required in the parent step are: (1) buffer allocation (at the beginning); (2) prework insertion; (3) replacement of the recursive kernel launch with buffer insertions; (4) insertion of the

required barrier synchronization primitive; and (5) postwork transformation. As shown in Figure 4, steps (1)-(3) are fairly mechanic; however, steps (4) and (5) require some consideration.

If the original kernel includes barrier synchronization between the recursive kernel launch and the postwork, such synchronization must be preserved in the consolidated kernel. For warp-level consolidation, this problem is automatically solved by the implicit barrier synchronization due to the lockstep execution of the threads within a warp. For block-level consolidation, CUDA provides a block-level barrier synchronization primitive (__syncthreads). Grid-level consolidation requires more thought. First, the only global barrier synchronization provided by CUDA is the implicit one at the end of a kernel launch. However, using this mechanism would require splitting the parent kernel into two: a kernel handling the prework and the child kernel launch, and a kernel handling the postwork. In addition, CPU intervention would be required to invoke the postwork-kernel. This would be problematic in case of recursion, as it would require returning the control to the CPU after each child kernel launch, and to have calls to the postwork-kernel stacked on CPU. In other words, the CPU would acquire full recursion control, leading to the overheads discussed in Chapter 2.2. To address this problem, I implement a custom global synchronization mechanism that can be invoked from the GPU (Chapter 4.2.5). Second, a global synchronization may cause a deadlock when active blocks on GPU are suspended at the global barrier while pending blocks are waiting for active blocks to finish. To address this problem, I consolidate the postwork into a single kernel. The last block to complete its buffer insertions will then launch the consolidated child kernel, wait for its completion and then launch the consolidated

postwork kernel. The other blocks will simply exit after completing their buffer insertions. Finally, the required barrier synchronization between the child kernel launch and the postwork is handled by inserting a *cudaDeviceSynchronize* call between the invocations of these two consolidated kernels. Dependencies between the prework and the postwork are handled by duplicating in the postwork the relevant portions of prework.

4.2.4. Consolidation Buffers

The design of the consolidation buffers involves several considerations, some of them leading to the need for the directive clauses listed in Table I.

Memory selection: The consolidation buffer can be either implemented in global or in shared memory. Global memory provides slower access but is visible to both parent and child kernels. Conversely, shared memory is faster but private to each block (and thus not visible within nested kernels). While parent kernels could fill temporary buffers in shared memory and then copy them into global memory to allow access by child kernels, the limited size of shared memory makes this solution not scalable. As a result, I store the consolidation buffers solely in global memory.

Dynamic allocation method: For the allocation of the consolidation buffers, I allow three alternatives: (1) the default allocator provided by CUDA; (2) the open-source halloc memory allocator for GPU [12]; and (3) a customized allocator that leverages a pre-allocated memory pool.

Buffer size for customized allocator: Due to the irregular nature of nested parallelism, the buffers required by different consolidated kernels may vary in size. When using the pre-allocated memory pool, the programmer needs to set both its size and the size of the portion of the memory pool allocated to each buffer (recall that in warp/block-level

consolidation every warp/block uses a consolidation buffer). The size of the pre-allocated memory pool (500MB by default) can be specified using the *totalSize* argument in the pragma directive. The per-buffer size (*perBufferSize*) is predicted as:

*totalThread * totalBuffVar * const*

where *totalThread* is the total number of threads from which I consolidate the child kernels, *totalBuffVar* is the number of buffered variables (indexes or pointers) per work item; and *const* is a constant (default value: 4) that estimates the number of work items assigned to a single thread. I have observed that, in most cases, the *perBufferSize* can be determined from a runtime variable that indicates a property of the current work item. For instance, for the tree applications in the benchmarks, the buffer size can be derived from the variable that indicates the number of children of a given node. If the user cannot provide such variable, a constant may also be specified to its best estimation. The customized allocator can utilize the information from the *#pragma* to allocate properly sized buffers from the pre-allocated global memory pool for different consolidation granularities. Notice that for grid level consolidation, only one buffer is required for each grid: in this case, the grid can directly utilize the whole memory pool and the *perBufferSize* is ignored.

4.2.5. Global Barrier Synchronization on GPU

The global barrier synchronization is implemented using a counter whose value is initialized to the number of blocks executed. When a block reaches the barrier, the first thread in the block decrements the counter by one atomically. A counter decrement to zero indicates that the last block has reached the barrier.

4.2.6. Kernel Configuration Handling

When launching kernel calls, it is common wisdom to select a configuration that achieves high device occupancy, which is defined as the ratio of the number of active warps to the number of maximum active warps that the device can host. Although higher occupancy does not always guarantee higher performance, it usually produces a good enough result that can be further tuned. The use of the *CUDA Occupancy Calculator* allows finding a kernel configuration (*B, T*) that maximizes the occupancy for a single kernel. However, concurrent kernels must share the GPU resources, and thus such configuration will not be optimal for concurrent kernels initiated with DP [11]. To allow multiple kernels to be concurrently active on GPU, programmers need to downgrade the configuration they obtain by using the *Occupancy Calculator*. I refer to Kernel Concurrency (*KC*) as the maximum number of concurrently active kernels. The highest *KC* supported by CUDA as of compute capability 3.5 is 32. Due to the hardware resource limitations, a concurrency of X may be achieved by downgrading the configuration (*B, T*) to *([B/X], T*). I name such configuration *KC_X*.

For grid-level consolidation, at any given time there is only one active consolidated kernel that processes all the work from all threads in the parent kernel. Hence, I expect the best configuration to be the one that maximizes the device occupancy for a single kernel. Thus, I use *KC_1* as the default kernel configuration.

For block-level consolidation, each block in the parent kernel will spawn a consolidated kernel that will handle a smaller amount of work collected from a block in the parent kernel. I decide to downgrade the configuration by a factor of 16 and use *KC_16* as the default configuration.

For warp-level consolidation, each warp in the parent kernel will spawn a consolidated kernel that handles an even smaller amount of work collected from a warp, and a maximum concurrency of 32 can be easily achieved. Thus, I use *KC_32* as the default kernel configuration.

Because users may need to fine-tune the configuration for their applications to achieve the best performance, I also provide *#pragma* clauses that allow for user-specified kernel configurations.

Chapter 5 Experimental Evaluation

5.1. Experimental Setup

Hardware and Software Setup: All experiments are performed on a workstation powered by two 6-core Xeon E5-2620 CPUs and an NVIDIA K20c GPU. The machine uses CentOS release 6.4 and CUDA runtime and compiler version 7.0. Nvidia Visual Profiler is used 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: The performance of the proposed methods were evaluated on three recursive applications--tree descendants, tree heights and BFS.

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.

Breadth First Search (BFS): For BFS, I used the block-mapped implementation described in [13] as code variant exhibiting flat parallelism. This implementation is work-efficient and traverses the graph level by level. The recursive code recurses over the neighbors of each node. Since the scheduling of parallel recursive calls is non-deterministic, in the 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.

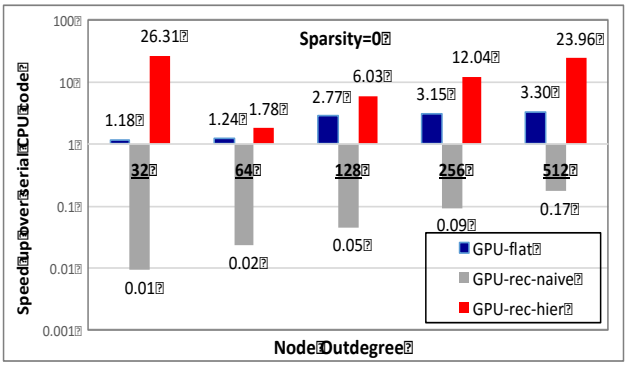
Datasets: For BFS, Kron_log16 from the DIMACS implementation challenges is used [14]. For Tree Descendants/Heights: synthetic datasets are generated. The tree generator produces trees with different shapes based on three parameters: *tree depth*,

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.

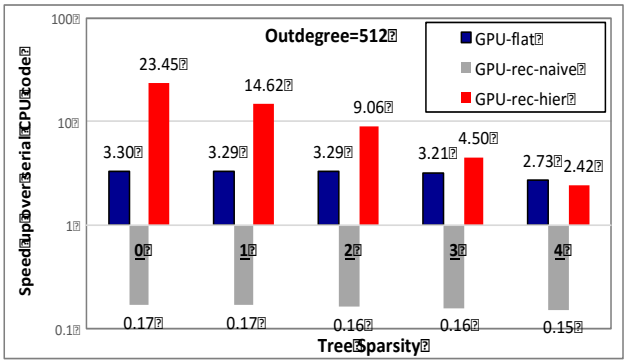
5.2. Experiments on Hierarchical Recursive Template

Results on Tree Descendants/Heights: I evaluate the performance of Tree Descendants and Tree Heights on synthetic datasets. The tree generator produces trees with different shapes based on three parameters: *tree depth*, *node outdegree* and *sparsity*. For both algorithms, I perform two sets of experiments over trees with different depths. First, I 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, I 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 5(a) and (b) show the results reported by the *Tree Descendants* algorithm on



(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 5 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 log10 scale: the values on top of the bars indicate the exact speedup numbers.

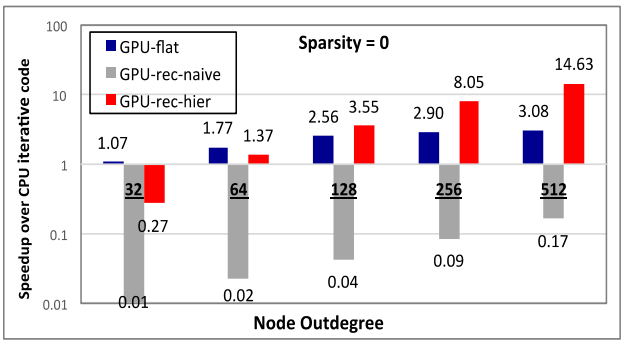
regular and irregular trees, respectively. Figure 5(c) shows the following profiling information for each code variant: warp utilization, number of dynamic kernel calls and number of atomic operations.

As can be seen in Figures 5(a) and (c), on regular trees the recursive-naïve implementation leads to significant performance degradation over the serial CPU code (and the 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.

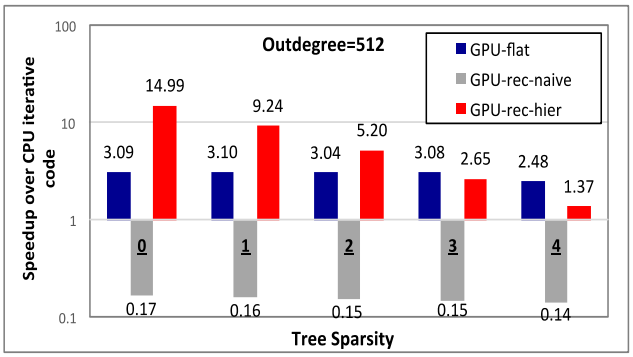
As can be seen in Figures 5(b) and (c), the sparsity parameter does not significantly affect the performance and warp utilization of the flat-parallelism code variant. On the other hand, the performance of the recursive-hierarchical code decreases as the sparsity increases. As can be seen in Figure 6(c), in this case the more irregular structure of the tree has a negative effect on the warp utilization, causing this performance degradation.

Figure 6 shows the performance and profiling results reported by the *Tree Heights* algorithm on the same synthetically generated trees. The tree height is defined 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



(b) 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 6 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 log10 scale: the values on top of the bars indicate the exact speedup numbers.

be seen in Figures 6(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 6(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.

5.3. Experiments on Workload Consolidation Schemes

Results in Chapter 5.2 shows that to make DP-based implementations beneficial, one should try to use few nested kernel invocations that spawn a significant amount of parallel work like the hierarchical recursive template for trees/graphs does. However, most of the basic DP-based implementations as shown in Figure 1 and Figure 2(d) will not satisfy this requirement. Although hierarchical recursive template can be preferable to flat implementations, its use is limited to trees and graph based applications.

5.3.1. Selection of the Kernel Configuration

In Chapter 4.2.6 three configurations are discussed for consolidated kernels: KC_1, KC_16 and KC_32. These configurations allow the consolidated kernels to achieve concurrency levels (maximum number of concurrently active kernels) of 1, 16 and 32, respectively. Now these configurations are compared with two additional configurations schemes, 1-1 mapping and exhaustive search. The 1-1 mapping configuration indicates that the kernel is configured to have as many blocks (or threads, in the case of thread-mapped child kernels [15]) as the number of items in the buffer. The exhaustive search configuration is the best configuration found from exhaustively searching the configuration space [16]. Figure 7 reports the results on tree descendants for all considered consolidation granularities over two datasets.

For the three proposed configurations. KC_1 works best for grid-level consolidation;

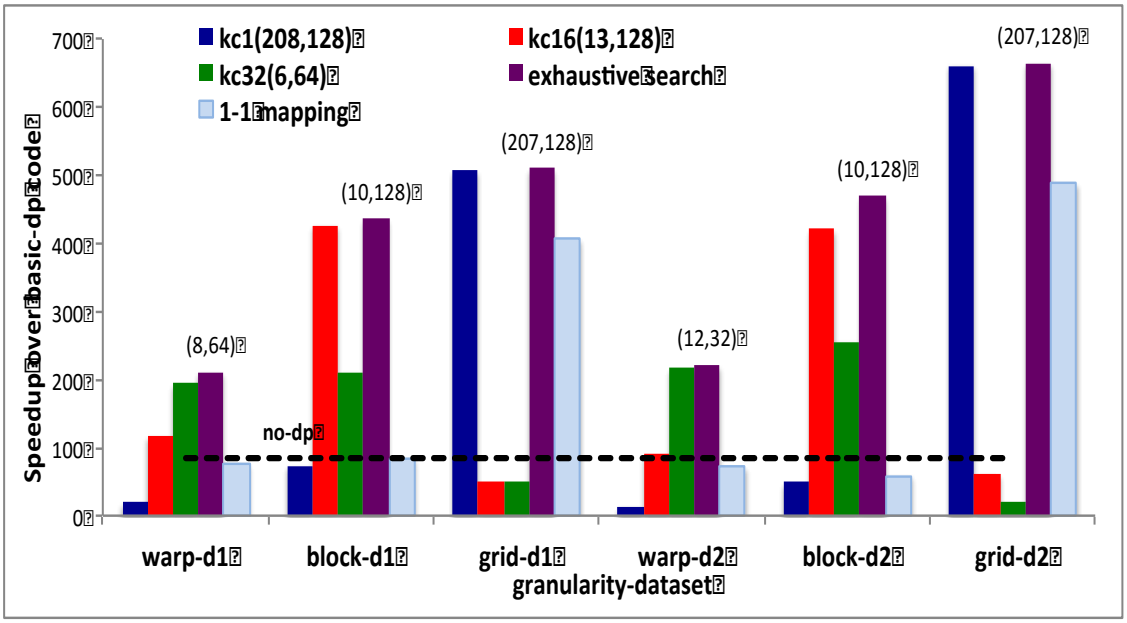


Figure 7 Performance of different kernel configurations (TD). All results are normalized to basic dynamic parallelism code.

KC_16 works best for block-level consolidation; and KC_32 works best for warp-level consolidation. This meets the expectations and is coherent with the analysis presented in Chapter 4.2.5. Then KC_1 for grid-level, KC_16 for block-level and KC_32 for warp-level consolidation are compared with 1-1 mapping. As can be seen, proposed solutions perform much better, especially for block- and warp-level consolidation. This is because the varying block size of 1-1 mapping lowers the Kernel Concurrency and increases the size of the pending queue, leading to higher overhead and degraded performance. At last, proposed schemes are compared with the best configuration found by exhaustive search. It can be seen that the configurations selected from the proposed methods (KC_1 for grid-level, KC_16 for block-level and KC_32 for warp-level consolidation) achieve on average 97% of the performance of the optimal configuration found by exhaustive search.

The same experiments conducted on the other benchmarks using various datasets report similar results. In conclusion, the proposed configuration selection method for

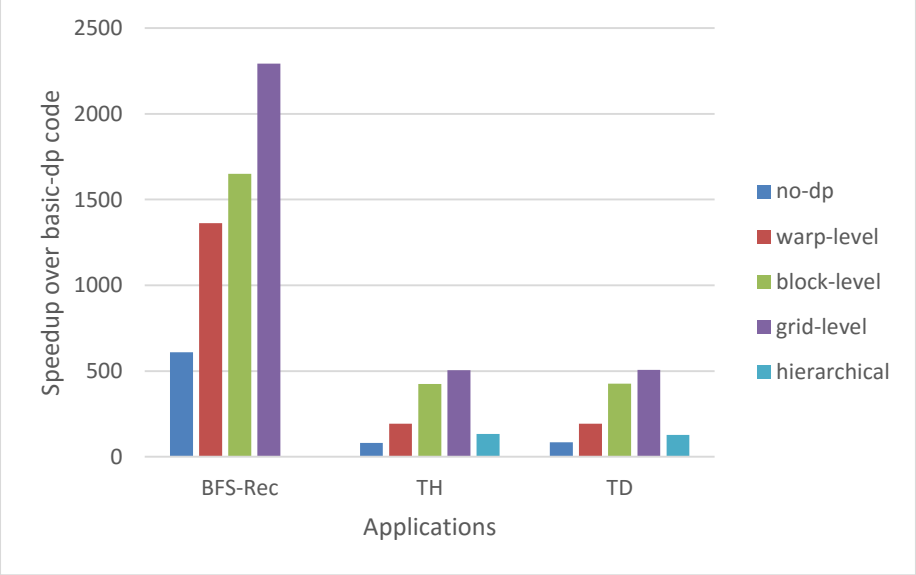


Figure 8 Overall speedup over basic dynamic parallelism.

nested kernels is effective and leads to nearly optimal performance. In all the remaining experiments, KC_1, KC_16 and KC_32 configurations are used for kernels consolidated at the grid-level, block-level and warp-level granularities, respectively.

5.3.2. Overall Performance

Figure 8 presents the overall speedup of kernels consolidated at different granularities over basic-dp. The chart also reports the speedup achieved by flat parallel code (no-dp) over the baseline and the speedup achieved by hierarchical template. As can be seen, the basic-dp implementation suffers from severe performance degradation due to the significant kernel management overhead and the limited GPU utilization. Even compared with the flat GPU kernels, basic-dp reports slowdown factors from 80 to 1100. Warp-level consolidation improves the performance of basic-dp on average by a factor of 1000x but in some cases is not significantly better than the flat GPU kernel (no-dp). Block-level consolidation outperforms warp-level consolidation, and grid-level consolidation achieves the best performance across all benchmarks. Even if warp-level consolidation has the benefit of very low synchronization overhead, when compared to block- and grid-level code, it suffers from the more significant overhead introduced by the additional child kernel launches. Grid-level consolidation reduces the number of child kernel launches dramatically, and thus achieves the best performance despite its extra synchronization overhead. On average, warp-level, block-level and grid-level consolidation outperform basic-dp by a factor of 999, 1357 and 1459, respectively, and no-dp by a factor of 2.18, 3.26 and 3.78, respectively.

5.3.3. Profiling Results

This section presents the analysis of the improvements in hardware utilization achieved

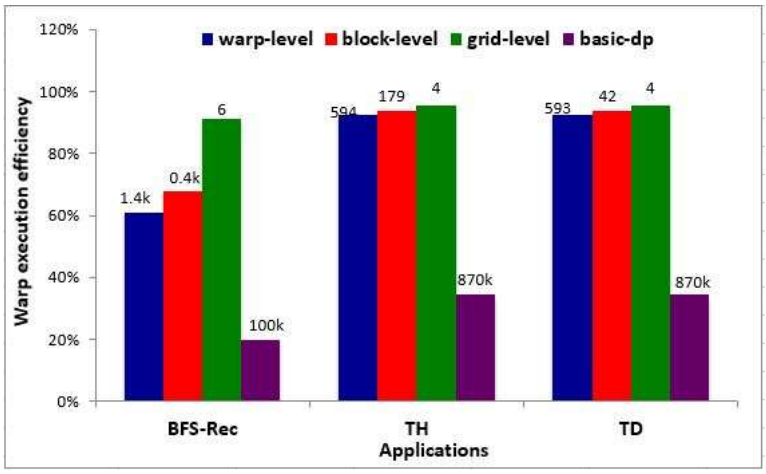


Figure 9 Warp execution efficiency across benchmarks.

by workload consolidations.

Figure 9 shows the overall warp execution efficiency, which is defined in the CUDA documentation [17] as "the ratio of the average active threads per warp to the maximum number threads per warp". For each application, results reported by the basic-dp

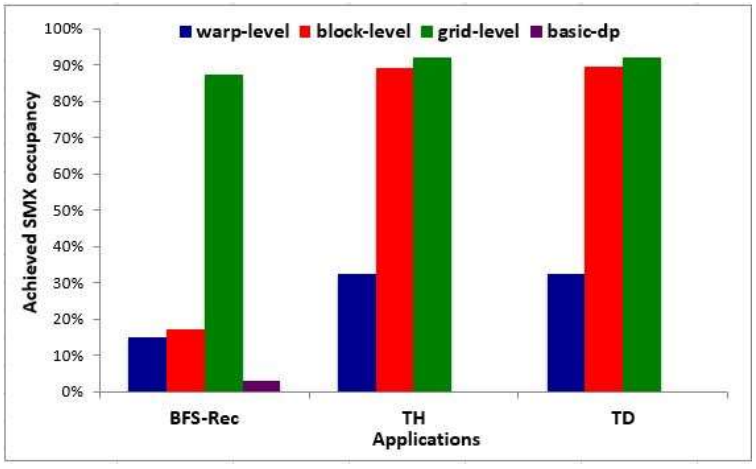


Figure 10 SMX occupancy (achieved hardware utilization). For TH and TD, the profiler reports "overflow" errors in basic-dp.

implementation and the three considered workload consolidation schemes are presented. On top of the bars is the number of child kernel invocations performed in each case. First, it can be seen that the proposed consolidation methods reduce the number of kernel invocations to 0.07%-14.48% of the ones performed by the corresponding basic-dp code. Second, average warp execution efficiencies are improved from 33.2% (basic-dp) to 69.3% (warp-level), 75% (block-level) and 83.1% (grid-level). The warp execution efficiency measured by Nvidia profiler includes not only parent and child kernels execution, but also child kernel launch overhead. Child kernel launches will take more clock cycles than buffer insertion operations, decreasing the warp efficiency. Consolidation replaces kernel launches with buffer insertions; as a result, it reduces the overhead of warp divergence and leads to improved overall warp efficiency.

Figure 10 shows the achieved streaming multiprocessor occupancy, which measures the ratio of average active warps over maximum warps supported per streaming multiprocessor [17]. On average, workload consolidation improves this metric from 27.9% (basic-dp) to 39.3% (warp-level), 60.3% (block-level), and 82.9% (grid-level). Recall that in basic-dp each thread launches a "small" kernel. Hence, the GPU device is filled with many such "small" concurrent kernels. As mentioned in Chapter3.2, there is a hardware limitation on the maximum number of concurrent kernels that a GPU can accommodate. On K20c, this limit is 32 with computing capability 35. Therefore, in the basic-dp case, thirty-two "small" concurrent kernels will underutilize the hardware. Workload consolidation, on the other hand, increases the average child kernel size and improves resource utilization.

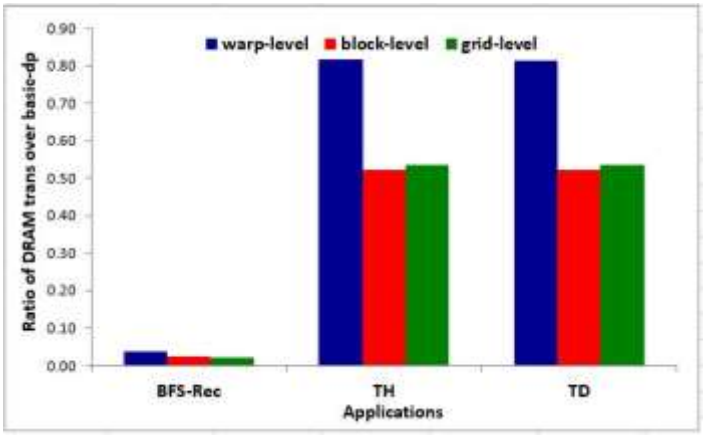


Figure 11 DRAM transactions ratio over basic dynamic parallelism.

To measure the efficiency of memory accesses, the numbers of DRAM transactions (read+write) performed by each kernel are monitored. Figure 11 shows the ratio between the number of DRAM transactions performed by each consolidated kernel and those performed by the basic-dp code. In all cases, the total DRAM transactions are reduced. Specifically, warp-level, block-level and grid-level consolidation lead to 60%, 34% and 36% of the original transactions, respectively. This reduction in memory transactions can be motivated as follows: first, the consolidation increases the average child kernel size, thus leading to better caching behavior and memory bandwidth utilization; second, a decrease in the number of nested kernels will lower the chance of swapping parent kernels out, therefore reducing the memory transaction overhead associated to kernel swapping; third, the decreased number of nested kernels reduces the use of the virtualized pool within the pending queue, lowering the overhead of virtual pool management.

Chapter 6 Conclusion

In this work, I propose two mechanisms – recursive parallelization templates and a workload consolidation scheme – to improve the efficiency of DP-based recursive codes. I evaluate and compare them with non-recursive based flat GPU codes and naïve DP codes.

Among the parallel recursive templates proposed, the use of DP-based hierarchical parallelization template on recursive tree traversal algorithms 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 (flat kernels). However, the benefits of DP on recursive applications operating on graph data structures are still unclear, especially when recursive code variants require synchronization through atomic operations.

The DP-based hierarchical recursive template can be preferable to flat kernels. However it can only be applied to tree and graph based applications. The workload consolidation technique is a more general solution that can significantly improve the performance of basic DP-based implementations by reducing the overhead of managing dynamic kernels and increasing the level of concurrency. My evaluation shows that kernels consolidated with the proposed code transformations achieve an average speedup in the order of 1500x over basic implementations using DP and an average speedup of 3.9x over optimized flat GPU kernels for both tree traversal and graph based applications.

Future Work

Currently, I am trying to automate the code transformation for the workload Consolidation by implementing a compiler. In order to direct the code transformations, I provide a single directive that can be applied to generic DP-based code following the template in Figure 1. This directive allows identifying the kernels to be consolidated and the work to be buffered. The grammar of the directive is: ***'#pragma dp [clause+]'***. Table 1 lists the clauses available, which specify the consolidation granularity, the type and size of the consolidation buffer, the indexes or pointers to the work items to be buffered and the configuration of the consolidated kernel. Some of these clauses are optional and programmers are supposed to use them for further optimization and performance tuning. For instance, developers can optionally specify the configuration of the consolidation buffers and the one of the consolidated kernels.

Table 1. Clauses of proposed workload consolidation compiler directive

Clause	Argument	Description	Optional
consldt	granularity: *warp*, *block*, *grid*	Workload consolidation granularity	No
buffer	type: *default*, *halloc*, *custom*	Buffer allocation mechanism	Yes
	perBufferSize: integer or variable name	Buffer size	
	totalSize: integer	Total size of all buffers	
work	varlist: list of indexes or pointers to work	List of variables to be stored in buffer	No
threads	thread number: integer	Number of thread/block for consolidated kernel	Yes
blocks	block number: integer	Number of blocks for consolidated kernel	Yes

Figure 12(a) illustrates the use of the proposed compiler directive to annotate the original CUDA code. In this case, block-level consolidation is selected, the buffer can

```
__global__ void rec_kernel() {
  work_item = get_work_item(…)
  prework(work_item)
  if (condition) {
  #pragma dp consldt(block) buffer(default, 256) work(work_item)
    rec_kernel<<<block_dim, thread_dim>>>(…, work_item, …)
  } else  work(work_item)
  postwork(work_item)
}
```

(a) Annotated CUDA code

```
__global__ void rec_kernel_consolidate() {
  buff_init();
  work_item = get_work_item_from_buff(…);
  prework(work_item)
  if (condition)  insert_buffer(work_item)
  else  work(work_item)
  synchronize
  if (thread_id==selected)
    rec_kernel_consolidate<<<b_dim_con, t_dim_con>>>()
  synchronize
  postwork(work_items_in_buff)
}
```

(b) Generated CUDA code

Figure 12 Example of use of the workload consolidation compiler directive: (a) original annotated code and (b) generated CUDA code. The generated CUDA code is generic (it applies to all consolidation schemes). For block-level consolidation, the *synchronize* primitive used is *__syncthreads*. For warp-level consolidation, the synchronization is implicit and no synchronization primitive is required. For grid-level consolidation, *synchronize* is a custom global synchronization primitive.

have at most 256 elements and is instantiated with the default CUDA allocator, and variable work_item is buffered. The compiler should generate the code shown in Figure 12(b) (in this particular case, the synchronization primitive in use is *__synchthread*).

To implement the compiler, I am using **Rose compiler infrastructure** (version 0.96.a), which incorporates Edison Design Group (EDG) Frontend 4.0 that supports the parsing of CUDA, C and C++ code [18]. Rose provides parser building APIs. I have already used them to implement the parser for the proposed pragma directive. Then I attached the pragma information to the annotated abstract syntax tree (AST) generated from the frontend parser. Based on the directive information and the AST, currently, I am writing the traversal and transformation functions to generate a new AST, which is then fed to and unparsed by the backend of ROSE to generate the consolidated recursive kernels.

References

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

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

[3] Y. Yang and H. Zhou, "CUDA-NP: realizing nested thread-level parallelism in GPGPU applications," presented at the PPoPP, 2014.

[4] 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, 2008.

[5] D. Li and M. Becchi, "Multiple Pairwise Sequence Alignments with the Needleman-Wunsch Algorithm on GPU," presented at the SC, Salt Lake City, UT, 2012.

[6] D. Li, K. Sajjapongse, H. Truong, G. Conant, and M. Becchi, "A Distributed CPU-GPU Framework for Pairwise Alignments on Large-Scale Sequence Datasets," presented at the ASAP, Ashburn, VA, 2013.

[7] J. Mosegaard and T. S. Sørensen, "Real-time deformation of detailed geometry based on mappings to a less detailed physical simulation on the gpu," presented at the Proceedings of the 11th Eurographics conference on Virtual Environments, 2005.

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

[9] G. E. Blelloch and G. W. Sabot, "Compiling collection-oriented languages onto massively parallel computers," *Parallel Distrib. Comput.,* vol. 8, pp. 119-134, 1990.

[10] M. Goldfarb, Y. Jo, and M. Kulkarni, "General transformations for GPU execution of tree traversals," presented at the Super Computing, 2013.

[11] V. T. Ravi, M. Becchi, G. Agrawal, and S. Chakradhar, "Supporting GPU sharing in cloud environments with a transparent runtime consolidation framework," presented at the HDPC, 2011.

[12] A. V. Adinetz and D. Pleiter. *Halloc: A fast and highly scalable GPU dynamic memory allocator*. Available: https://github.com/canonizer/halloc

[13] P. Harish and P. J. Narayanan, "Accelerating large graph algorithms on the GPU using CUDA," presented at the HiPC, 2007.

[14] *DIMACS Implementation Challenges*. Available: http://dimacs.rutgers.edu/Challenges/

[15] D. Li and M. Becchi, "Deploying Graph Algorithms on GPUs: an Adaptive Solution," presented at the IPDPS, Boston, MA, 2013.

[16] S. Muralidharan, M. Shantharam, M. Hall, M. Garland, and B. Catanzaro, "Nitro: A Framework for Adaptive Code Variant Tuning," presented at the IPDPS, 2014.
[17] *Profiler User's Guide*. Available: http://docs.nvidia.com/cuda/profiler-users-guide/
[18] D. J. Quinlan, C. Liao, J. Too, R. P. Matzke, and M. Schordan. (2015). *ROSE Compiler Infrastructure*. Available: http://www.rosecompiler.org