KLAP: Kernel Launch Aggregation and Promotion for Optimizing Dynamic Parallelism

Izzat El Hajj*‡, Juan Gómez-Luna†, Cheng Li*, Li-Wen Chang*, Dejan Milojicic‡ and Wen-mei Hwu*
*University of Illinois at Urbana-Champaign
†Universidad de Córdoba
‡Hewlett-Packard Labs
elhajj2@illinois.edu, el1goluj@uco.es, {cli99, lchang20}@illinois.edu, dejan.milojicic@hpe.com, w-hwu@illinois.edu

***Abstract*—Dynamic parallelism on GPUs simplifies the programming of many classes of applications that generate parallelizable work not known prior to execution. However, modern GPUs architectures do not support dynamic parallelism efficiently due to the high kernel launch overhead, limited number of simultaneous kernels, and limited depth of dynamic calls a device can support.**

In this paper, we propose Kernel Launch Aggregation and Promotion (KLAP), a set of compiler techniques that improve the performance of kernels which use dynamic parallelism. Kernel launch aggregation fuses kernels launched by threads in the same warp, block, or kernel into a single aggregated kernel, thereby reducing the total number of kernels spawned and increasing the amount of work per kernel to improve occupancy. Kernel launch promotion enables early launch of child kernels to extract more parallelism between parents and children, and to aggregate kernel launches across generations mitigating the problem of limited depth.

We implement our techniques in a real compiler and show that kernel launch aggregation obtains a geometric mean speedup of 6.58× over regular dynamic parallelism. We also show that kernel launch promotion enables cases that were not originally possible, improving throughput by a geometric mean of 30.44×.

I. Introduction

Many modern GPUs come with support for dynamic parallelism. *Dynamic parallelism* [1], [2] is the ability of a kernel running on a GPU to spawn child kernels from the GPU without returning to the host. This feature makes it easier to program many classes of applications that dynamically generate variable amounts of parallel work not known prior to execution. Such applications include graph traversal [3], mesh refinement [4], and other kinds of algorithms. Dynamic parallelism also simplifies the programming of applications with complex inter-block dependence such as producer-consumer algorithms [5].

Although dynamic parallelism improves developer productivity and code maintainability [3], [6], [7], [8], current hardware support for it can be very inefficient in practice. One limitation of the current hardware in supporting dynamic parallelism efficiently is the high overhead of launching subkernels from the device [6], [7]. Another limitation is that the number of kernels that can be in flight at a time is limited [6], [9]. The effect of both these limitations is exacerbated when many threads of a parent kernel each launch a small child kernel. In this case, the many subkernel launches will incur the launch overhead multiple times, and the small granularity of the kernels will result in low occupancy, underutilizing the GPU resources. Yet another limitation of dynamic parallelism is the bound on the depth of the call stack, which is problematic for computation patterns with high amounts of recursion and long dependence chains [6] such as producer-consumer algorithms.

Hardware and software approaches have been proposed for improving the performance of code that uses dynamic parallelism on GPUs. Dynamic thread block launch [9] proposes architectural changes that enable kernels to dynamically launch lightweight thread blocks. Free launch [10] is a software approach that eliminates subkernel launches entirely by reusing parent threads on the GPU to perform the work of child kernels.

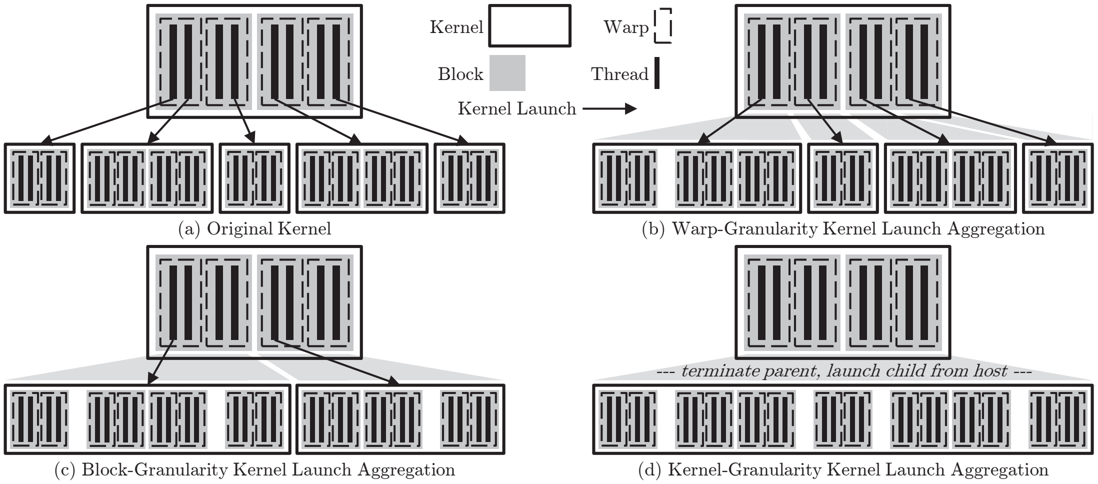
In this paper, we propose Kernel Launch Aggregation and Promotion (KLAP), a set of compiler techniques that improve the performance of kernels using dynamic parallelism on modern GPUs. KLAP applies kernel launch *aggregation* to fuse kernels launched by threads in the same warp, block, or kernel together into a single launch. Aggregation thus reduces many fine-grain kernels into fewer coarser-grain ones, thereby incurring fewer launches and allowing more work to be scheduled simultaneously for better occupancy. For producer-consumer computation patterns, KLAP employs kernel launch *promotion* whereby kernels are launched by the parent prematurely. Promotion enables overlapping the independent part of the child kernel with its parent and also aggregating kernel launches across multiple descendants which mitigates the problem of limited depth.

KLAP does not require any new architecture support and is therefore compatible with current GPUs that support dynamic parallelism. Moreover, KLAP does not reuse parent threads nor does it eliminate dynamic kernel launches entirely, but rather uses the dynamic kernel launch capability in a more efficient manner. Leveraging the dynamic parallelism capability affords KLAP more flexibility and generality than can be achieved by relying entirely on thread reuse which suffers the limitations associated with persistent threads.

We make the following contributions:

- We propose kernel launch aggregation, a novel compiler technique which improves performance of dynamic

978-1-5090-3508-3/16/$31.00 © 2016 IEEE



(a) Original Kernel

(b) Warp-Granularity Kernel Launch Aggregation

(c) Block-Granularity Kernel Launch Aggregation

(d) Kernel-Granularity Kernel Launch Aggregation

Fig. 1. Kernel Launch Aggregation

parallelism on GPUs by reducing the number of kernel launches and improving occupancy.

- We propose kernel launch promotion, a compiler transformation that enables work overlap and kernel launch aggregation and mitigates the problem of limited depth for producer-consumer computation patterns.
- We implement our techniques in a real compiler and evaluate its performance on real hardware for multiple GPU architectures supporting dynamic parallelism.
- We show that kernel launch aggregation obtains a geometric mean speedup of 6.58× over regular dynamic parallelism, and that kernel launch promotion enables cases that were not originally possible, improving throughput by a geometric mean of 30.44×.

The rest of this paper is organized as follows: Section II describes kernel launch aggregation, Section III describes kernel launch promotion, Section IV evaluates our techniques, Section V outlines related work, and Section VI concludes.

II. Kernel Launch Aggregation

Kernel launch aggregation is a transformation whereby kernels that were originally launched by multiple threads are aggregated into a single kernel which is launched once. The *granularity* of aggregation is the scope of threads across which the kernel launches are aggregated. For example, kernel launch aggregation at warp granularity means that kernels launched by threads in the same warp are aggregated into a single kernel which is launched by one of the threads in that warp; on the other hand, kernels launched by threads in different warps remain separate.

In this paper, kernel launch aggregation is done at three different granularities: warp, block, and kernel. Aggregation at warp and block granularity is described in Section II-A while aggregation at kernel granularity is described in Section II-B.

A. Warp and Block Granularity

Figure 1(b) illustrates the transformation that takes place when kernel launch aggregation at warp granularity is applied to the example in Figure 1(a). In this toy example, the first warp in the parent kernel originally had two threads each launching a child kernel. In the transformed version, the two child kernels are aggregated into the same kernel which is launched by one of the two threads in the parent warp. This transformation effectively reduces the number of kernel launches by up to a factor of the warp size.

The transformation at block granularity illustrated in Figure 1(c) is very similar. Here, only one thread per block launches a kernel on behalf of all the threads in the block. Thus, the number of kernel launches is effectively reduced by up to a factor of the block size.

The code transformation to perform warp (or block) granularity aggregation is shown in Figure 2 and an example of what this code does is shown in Figure 3. Pseudocode is used and handling of corner cases is omitted for brevity and clarity. For readers interested in specifics, detailed code is shown in Figure 12 at the end of the paper.

Figure 2(c) shows how kernel calls inside kernel functions are transformed from that in Figure 2(a) to call an aggregated kernel. The first step in the transformed code is for the warp (or block) to allocate global arrays to store the arguments and configurations to be passed to the aggregated kernel (line 05). These arrays are needed because different parent threads may pass different arguments and configurations to their child kernels, therefore each thread must store its passed values in global arrays, and these arrays are passed to the aggregated child instead. Next, each thread stores its arguments and configurations in the allocated arrays (lines 06-07). The sum of the number of blocks in all the children is calculated as the new number of blocks in the aggregated kernel (line 08). Likewise, the maximum number of threads per block in all the children is calculated as the new number of threads in the aggregated kernel (line 09) to make sure all blocks in the aggregated child kernel have enough threads. Thus, our technique does not assume that the number of blocks and threads in the launched kernels are known at compile time or that they are

```
01  kernel<<<gD,bD>>>(args)
```

(a) Original Kernel Call

```
02  __global__ void kernel(params) {
03      kernel body
04  }
```

(b) Original Kernel

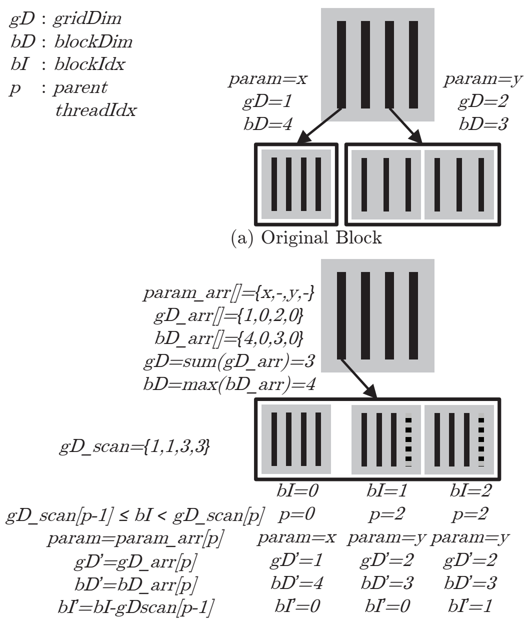
```
05  allocate arrays for args, gD, and bD
06  store args in arg arrays
07  store gD in gD array, and bD in bD array
08  new gD = sum of gD array across warp/block
09  new bD = max of bD array across warp/block
10  if(threadIdx == launcher thread in warp/block) {
11      kernel_agg<<<new gD, new bD>>>
12                          (arg arrays, gD array, bD array)
13  }
```

(c) Transformed Kernel Call (called in a kernel)

```
14  __global__ void kernel_agg(param arrays, gD array, bD array) {
15      calculate index of parent thread
16      load params from param arrays
17      load actual gridDim/blockDim from gD/bD arrays
18      calculate actual blockIdx
19      if(threadIdx < actual blockDim) {
20          kernel body  (with kernel launches transformed and with
21                         using actual gridDim/blockDim/blockIdx)
22      }
23  }
```

(d) Transformed Kernel (called from a kernel)

Fig. 2. Code Generation for Aggregation at Warp and Block Granularity



(a) Original Block

(b) Block-Granularity Aggregation Logic Example

Fig. 3. Aggregation Example

uniform across parent threads. Finally, one of the threads in the warp (or block) launches a single aggregated kernel on behalf of the others (line 10). For block granularity, a barrier synchronization is needed before the launch to ensure that all the threads in the block have completed their preparation of the arguments and configurations. In the aggregated kernel launch, the new configurations are used (line 11), arguments are replaced with argument arrays, and arrays containing the configurations for each original child are added (line 12).

In addition to transforming kernel launches in all original kernels, an aggregated version of each original kernel must also be created. Figure 2(d) shows how the kernel in Figure 2(b) is transformed into an aggregated version. First, all parameters are converted into parameter (param) arrays and configuration arrays are appended to the parameter list (line 14). Next, before the kernel body, logic is added for the block to identify which thread in the parent warp (or block) was its original parent (line 15). After identifying its original parent, the block is then able to load its actual configurations and parameters (lines 16-18). Threads that were not in the original child kernel are then masked out (line 19). Finally, in the kernel body, all kernel launches are transformed into aggregated kernel launches, and all uses of `blockDim` and `blockIdx` are replaced with the actual values (lines 20-21).

For the block to identify its original parent, it needs to execute a scan (prefix sum) on the `gD` (gridDim) array then search for its position (given by the aggregated `blockIdx` value) between the scanned values (using p-ary search [11]). In practice, since all child blocks need to scan the same `gD` array, the scan is instead performed once by the parent before the array is passed to the aggregated child kernel. Conveniently, the scan can be performed along with the preparation of the configuration and parameter arrays in the parent, making it incur little additional overhead. Since the child kernel needs both the scan value and the original `gD` value, it can recover the original `gD` value by subtracting adjacent scan elements. The scan is performed using CUB [12].

The transformed code requires that all threads are active to perform the scan and max operations. To handle control divergence, a preprocessing pass performs control-flow-to-data-flow conversion to convert divergent launches to non-divergent predicated launches so that all threads reach the launch point. Predication is achieved by multiplying the predicate with the grid dimension such that launches by inactive threads become launches of zero blocks.

B. Kernel Granularity

Figure 1(d) illustrates the transformation that takes place when kernel launch aggregation is applied at kernel granularity. At this granularity, all the original child kernels are aggregated into a single kernel. Because there is no global synchronization on the GPU, a single thread cannot be chosen to launch the kernel on behalf of the others once the others are ready. Instead, the child kernels are postponed and launched from the host after the parent kernel terminates. In order to postpone the kernel launches, this transformation requires that parent kernels do not explicitly synchronize with their child kernels, so kernels with explicit synchronization are not supported at this granularity.

The code transformation for kernel granularity aggregation is omitted for brevity. Compared with warp and block granularity aggregation, it has two main differences. The first difference is that after the parent kernel has performed all the setup operations for the aggregated child kernel call, the call will not take place. Instead, the aggregated child will be called from the host function after the parent kernel has returned. This effectively enforces a barrier synchronization among all parents in the grid before launching the aggregated kernel.

The second difference is in computing the aggregated kernel configurations. At the warp and block granularity, a regular tree-based scan is used. However, a tree-based scan at kernel granularity has two limitations. First, it would require additional kernels to be launched between the parent and the child to perform the scan. Second, it will be inefficient due to the potentially large number of zero values from threads that don't perform a launch (while these zeros exist at the other granularities, their overhead is not as large). For this reason, we employ a sequential out of order scan using `atomicAdd` which does not store zeros and can be performed in the parent kernel directly.

One challenge for kernel-granularity aggregation is when child-kernel launches in the parent kernel are contained in loops. In this case, the number of launches of each child kernel must be tracked and passed to the host so the host can launch the right number of children after the parent kernel terminates. Our implementation currently does not support this case, but it is technically feasible and the subject of future work.

C. Optimizations

This subsection discusses some optimizations that we perform to improve the efficiency of the generated code. One source of inefficiency is performing dynamic memory allocations in the kernel. To avoid such allocations when creating global arrays for storing parameters and configurations, we instead allocate a single global memory pool from the host code and use atomic operations to grab memory from that pool instead of calling `cudaMalloc` in the kernel.

A related optimization is aggregating calls to `cudaMalloc` that were part of the original code. At warp (or block) granularity, we transform calls to `cudaMalloc` by each thread in the warp (or block) into code that: (1) sums up the total allocated memory by the warp (or block), (2) uses one thread in the warp (or block) to allocate that total memory on behalf of the others, then (3) redistributes the allocated memory to all threads. At kernel granularity, `cudaMalloc` cannot be aggregated because it is a blocking call. In this case, we aggregate `cudaMalloc` at block-granularity instead.

Another optimization is avoiding the overhead of creating arrays for arguments that are uniform across the granularity of aggregation. For example, if we are performing block-granularity aggregation, and the compiler can prove that an argument has the same value for all threads in the block [13], [14], then an array does not need to be created for that argument. Instead the argument is passed as is to the aggregated child kernel as a single value.

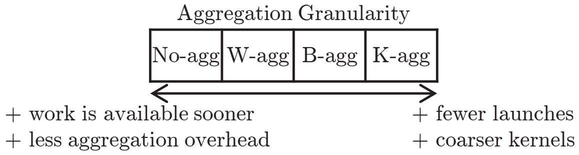


Fig. 4. Aggregation Granularity Tradeoffs

D. Tradeoffs

There are various advantages and disadvantages of increasing the granularity of aggregation as shown in Figure 4. With coarser-grain aggregation, there are fewer kernel launches and more work per kernel. Therefore, the kernel launch overhead is amortized over a larger amount of work and resources are better utilized. On the other hand, with finer-grain aggregation, threads launching aggregated kernels have to wait for fewer threads before performing the launch, making work from the aggregated kernel available sooner to utilize the GPU. Moreover, with finer-grain aggregation, there is less overhead from the aggregation logic because the scan operation to compute launch configurations are on a smaller scale, and the configuration arrays to be searched to identify parent threads are also smaller.

III. Kernel Launch Promotion

Kernel launch promotion applies to a common class of producer-consumer algorithms whereby each kernel contains a single block and calls itself recursively as shown in Figure 5(a). This class of algorithms is common in applications having complex inter-block dependence [5], [15].

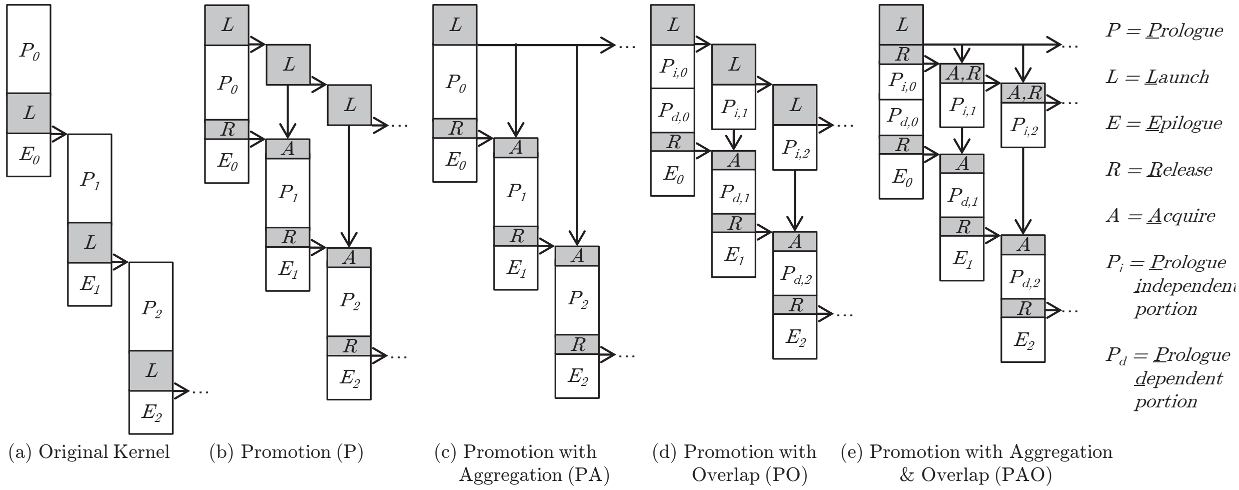
Implementing these patterns using dynamic parallelism greatly simplifies the expression of inter-block dependence, but has several limitations. These limitations are: (1) many fine-grain kernel launches, (2) a deep kernel call stack, and (3) a long serial dependence chain of single-block kernels. To address these problems, this section proposes kernel launch promotion.

Kernel launch promotion is a transformation whereby kernel calls are promoted to the beginning of a kernel to launch child kernels prematurely. Ordering between parent and child is then enforced via release-acquire synchronization. Promotion also enables two mutually orthogonal optimizations: aggregation and overlap. The benefit of aggregation is that it reduces the number of kernels and increases their granularity, as well as reduces the depth of the kernel call stack. The benefit of overlap is that it extracts more parallelism from the long serial dependence chain. These benefits are summarized in Figure 6.

The following subsections detail basic promotion (Section III-A) and promotion with aggregation (Section III-B), overlap (Section III-C), and both together (Section III-D).

A. Basic Promotion

Basic promotion is the transformation where kernel calls are hoisted to the beginning of the kernel. Figure 5(b) illustrates the transformation that takes place when basic promotion is



(a) Original Kernel (b) Promotion (P) (c) Promotion with Aggregation (PA) (d) Promotion with Overlap (PO) (e) Promotion with Aggregation & Overlap (PAO)

Fig. 5. Kernel Launch Promotion

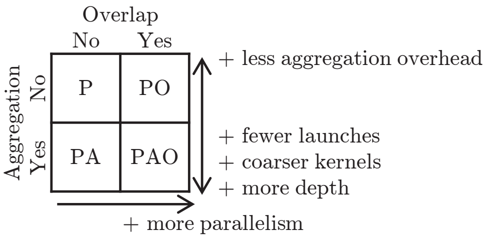


Fig. 6. Tradeoffs of Optimizations Enabled by Promotion

applied to the example shown in Figure 5(a). In the original kernel, the kernel call divides the kernel into two different sections called the *prologue* and the *epilogue*. In the transformed code, the kernel call is moved to the beginning of the kernel before the prologue. In place of the original kernel call, logic is inserted to *release* a flag that the child will *acquire* to know when to execute. In all but the first kernel instance (which is launched from the host), logic is also inserted before the prologue to acquire the release from the parent.

The code transformation to perform basic promotion is shown in Figure 7. Pseudocode is used and handling of corner cases is omitted for brevity and clarity. Figure 7(b) shows the transformation of the kernel in Figure 7(a) for the first kernel instance which is launched from the host. In the transformation, a distinction is made between the arguments which are available at the beginning of the kernel and arguments that are not available and need to be postponed. For the arguments that are not available, the launcher thread allocates buffers (line 10) and passes them to the child where the arguments will later be stored. A flag is also allocated for the parent to communicate with the child to release it (line 11). The child is then launched prematurely (lines 12-13).

After the prologue, the launcher thread stores the postponed arguments in the buffers (line 17) and executes a memory fence (line 18) to make sure all data is visible to the child before it is released. The launcher thread then sets the flag to release the child (line 19). In this paper, release is implemented in CUDA as a non-cached store while acquire is implemented as a polling loop with a volatile load. In OpenCL 2.0 [16], the built in support for release and acquire can also be used.

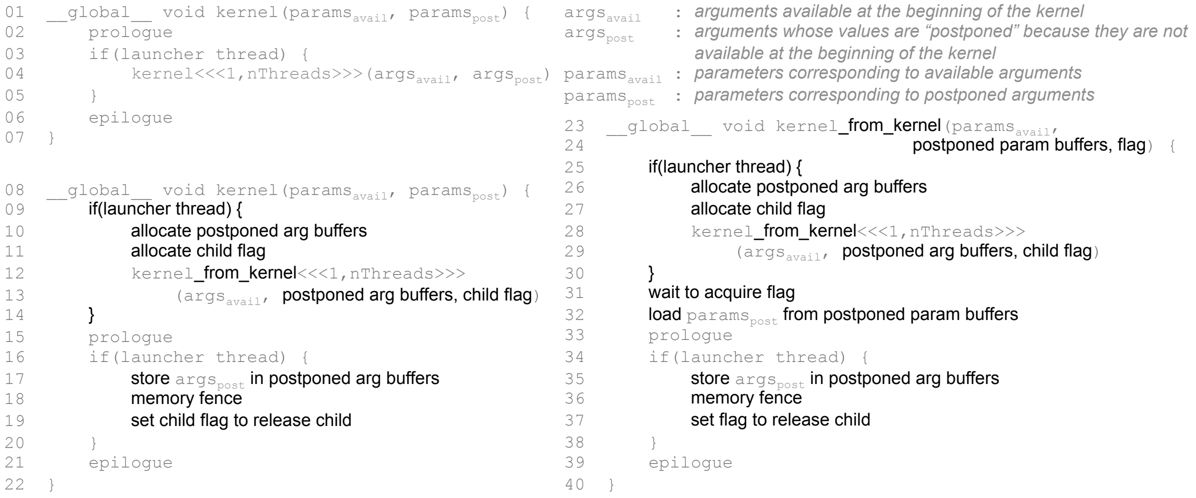
Figure 7(c) shows the transformation of the kernel in Figure 7(a) for the remaining kernel instances which are launched by device kernels. There are two main differences between this code and that of the first instance in Figure 7(b). The first difference is that instead of values for the postponed parameters being passed to the kernel, parameter buffers are passed as well as the flag (line 24). The second difference is that before the prologue, code is inserted to acquire the flag then load the postponed parameter values from the buffers (lines 31-32).

The transformation described requires the launch condition (not shown in the figure) to be promotable with the kernel launch. In cases where the condition is dependent on the prologue, launches can be performed speculatively and an abort flag can be used to abort the trailing launches.

B. Promotion with Aggregation

Figure 5(c) illustrates the transformation that takes place when aggregation is applied to the promoted kernel launch. In this transformation, instead of each parent launching its direct child as with basic promotion, a parent launches a large pool of descendants and creates parameter buffers and flags for each of these descendants. The size of this pool defines the granularity of aggregation. The descendants communicate via the flags to release each other one after the other. The block index within the pool of descendants is acquired dynamically [17], thus avoiding deadlock situations where child blocks get scheduled before their parents and starve them. If the pool of descendants is exhausted, the last descendant in the pool launches a new pool. When the final descendant is reached, it sets a bit in the flag for the remaining unused blocks in the pool to abort.

We note that aggregation in this context is different from the aggregation described in Section II. The aggregation in



```
__global__ void kernel(params_avail, params_post) {
    prologue
    if(launcher thread) {
        kernel<<<1,nThreads>>>(args_avail, args_post)
    }
    epilogue
}

__global__ void kernel(params_avail, params_post) {
    if(launcher thread) {
        allocate postponed arg buffers
        allocate child flag
        kernel_from_kernel<<<1,nThreads>>>
            (args_avail, postponed arg buffers, child flag)
    }
    prologue
    if(launcher thread) {
        store args_post in postponed arg buffers
        memory fence
        set child flag to release child
    }
    epilogue
}
```

```
args_avail    : arguments available at the beginning of the kernel
args_post     : arguments whose values are "postponed" because they are not
                available at the beginning of the kernel
params_avail  : parameters corresponding to available arguments
params_post   : parameters corresponding to postponed arguments
```

```
__global__ void kernel_from_kernel(params_avail,
                                   postponed param buffers, flag) {
    if(launcher thread) {
        allocate postponed arg buffers
        allocate child flag
        kernel_from_kernel<<<1,nThreads>>>
            (args_avail, postponed arg buffers, child flag)
    }
    wait to acquire flag
    load params_post from postponed param buffers
    prologue
    if(launcher thread) {
        store args_post in postponed arg buffers
        memory fence
        set flag to release child
    }
    epilogue
}
```

Fig. 7. Code Generation for Basic Promotion

Section II is *horizontal*, meaning that it aggregates kernel launches by threads in the same kernel at the same level of depth in the call stack. On the other hand, aggregation in this context is *vertical*, meaning that it aggregates kernel launches across multiple levels of depth in the call stack.

The benefits of promotion with aggregation are twofold. First, the number of kernel calls is reduced and their granularity is increased which results in better amortization of the launch overhead and better utilization of the device. Second, the architectural limitation of the depth of descent is mitigated because vertical aggregation divides the depth of the kernel call stack by a factor equal to the aggregation granularity (size of the pool in the aggregated kernel).

C. Promotion with Overlap

Promotion with overlap is based on the observation that portions of the prologue of a child kernel can be executed independently from the parent. Informally, this independent portion must satisfy two conditions: (1) it must not use any postponed parameters and (2) it must not have any data dependence with the prologue of the parent. If these two conditions are met, then this portion of the prologue can be executed in parallel with the parent before it releases the child.

Figure 5(d) illustrates the transformation that takes place when overlap is applied with promotion. In this transformation, the prologue is divided into two regions – the independent region (P_i) and dependent region (P_d) – and the independent region is hoisted before the acquire logic. Here, P_i and P_d must satisfy the conditions that $P_{i,x}$ and $P_{d,x}$ do not write to any memory referenced by $P_{i,y}$ where $x < y$, and $P_{i,y}$ does not write to any memory referenced by $P_{i,x}$ and $P_{d,x}$ where $x < y$. In this paper, a simple programmer annotation is used to indicate the boundary between P_i and P_d. However, dataflow analysis can also be used to detect these regions and is the subject of future work.

D. Promotion with Aggregation and Overlap

Figure 5(e) illustrates the transformation that takes place when both aggregation and overlap are applied with promotion. The two optimizations are orthogonal and interoperate nicely without much added complexity. The most noteworthy difference is that the transformed kernel now has two release-acquire chains. The first chain enforces the launch condition, thus ensuring that only the thread blocks that are supposed to execute do, while the trailing ones abort. The second chain enforces the dependence between parents and children.

IV. Evaluation

A. Methodology and Implementation Details

We implement our compiler in Clang version 3.8.0 as a source-to-source (CUDA-to-CUDA) translator. Although this version does not compile code that uses dynamic parallelism to LLVM IR, we modify the semantic checker to accept kernel calls inside kernel functions to enable the source-to-source transformation to take place.

The benchmarks and datasets used in the evaluation are shown in Table I. Aggregation and promotion are evaluated on different benchmarks because they target different patterns. For the aggregation benchmarks, a few (*bt*, *ccl*, *qt*) employ CUDA Dynamic Parallelism (CDP) originally, while the rest were ported to use CDP from original codes with intra-thread nested loops. For example, in *bfs*, the loop over the edges of a node is converted to a kernel launch with one thread processing each edge. For the promotion benchmarks, we chose algorithms which require communication between adjacent thread blocks, and implemented them using CDP. Each promotion benchmark is tested on three datasets: small which is selected to create 2 recurrences, medium which is selected to create 25 recurrences (the maximum CDP can handle), and large which is the maximum problem size the device can handle or the maximum recurrence that aggregation at granularity 128 can handle (whichever maxes out first).

Aggregation

Name	Description	Dataset	Thread Block Sizes
bfs	Breadth First Search [18]	Random, 10000 nodes, 1000 degree	parent=1024, child=32
bh	Barnes Hut Tree [19]	4096 bodies, 4 time-steps	parent=256, child=256
bt	Bezier Lines Tessellation [20]	25600 lines	parent=64, child=32
ccl	Connected Component Labelling [21]	8 frames, 4 host streams	parent=2, child=256
gc	Graph Coloring [22]	1 4096 0.01 (bcsstk13.mtx [23])	parent=256, child=256
mstf	Minimum Spanning Tree (find) [19]	rmat12.sym.gr [19]	parent=1024, child=1024
mstv	Minimum Spanning Tree (verify) [19]	rmat12.sym.gr [19]	parent=1024, child=1024
qt	Quadtree [20]	40000 points, 12 depth, 1 min.node	parent=128, child=128
sp	Survey Propagation [19]	random-42000-10000-3.cnf [19], 10000 literals	parent=384, child=64
sssp	Single-Source Shortest Path [19]	rmat12.sym.gr [19]	parent=128, child=128

Promotion

Name	Description	Dataset	Thread Block Sizes
los	Line of Sight [20]	small=511, medium=6399, large=49407	parent=256, child=256
pd	Padding [5]	small=120×120, medium=450×450, large=4600×4600	parent=512, child=512
pt	Partition [24]	small=16384, medium=204800, large=25174016	parent=512, child=512
sc	Stream Compaction [15]	small=16384, medium=204800, large=25174016	parent=512, child=512
unq	Unique [24]	small=16384, medium=204800, large=25174016	parent=512, child=512
upd	Unpadding [5]	small=120×120, medium=450×450, large=4600×4600	parent=512, child=512

TABLE I
BENCHMARKS

We run our experiments on both Kepler and Maxwell architectures. The Kepler GPU is an NVIDIA Tesla K40c coupled with an 8-core Intel Core i7 920 (2.67 GHz). The Maxwell GPU is an NVIDIA GeForce GTX 980 coupled with a 4-core Intel Core i3 530 (2.93 GHz). Both machines use CUDA SDK 7.5.

We enable per-thread default streams [25] for all benchmarks to allow kernels launched by threads in the same block to execute in parallel (default semantic is to serialize them). Enabling per-thread default streams is needed for the benchmarks to be amenable to aggregation, otherwise the serialization semantic would be violated. In addition, enabling default streams is good for the baseline CDP versions because it makes them faster (geomean 1.90× on Kepler and 1.83× on Maxwell). Further, we observe that using private streams is sometimes faster than per-thread default streams, so when that is the case, private streams are used in the baseline instead.

We use `cudaDeviceSetLimit` to adjust the fixed-size pool of the pending launch buffer appropriately for the baseline CDP version. Without the right size, the cost of overflowing the launch buffer pool would greatly penalize the execution time [26].

For the profiling results in Figure 9, we obtain the execution time breakdowns by incrementally deactivating parts of the code and measuring the resulting time difference. We deactivate code regions using conditionals that are always false but that cannot be proven so by the compiler, thus preventing the possibility of dead code elimination in the active regions. For iterative kernels with data-dependent convergence criteria (*bfs*, *mstf*, *mstv*, *sp*, *sssp*), we only profile the longest-running iteration because deactivating code of one iteration changes the behavior of later iterations. Likewise, for recursive kernels (*qt*), we only profile the longest running recurrence.

The results in Figure 10 are presented using throughput for each benchmark. All of them use effective memory throughput (GB/s) except *los* which uses ray length per second. The reason we use throughput is to make the numbers for small, medium, and large datasets comparable since the large dataset does not have a CDP baseline because CDP does not work.

For the profiling results in Figure 11, we use performance counters from the CUDA Profiler [27] to measure achieved occupancy and executed instruction count.

B. Aggregation

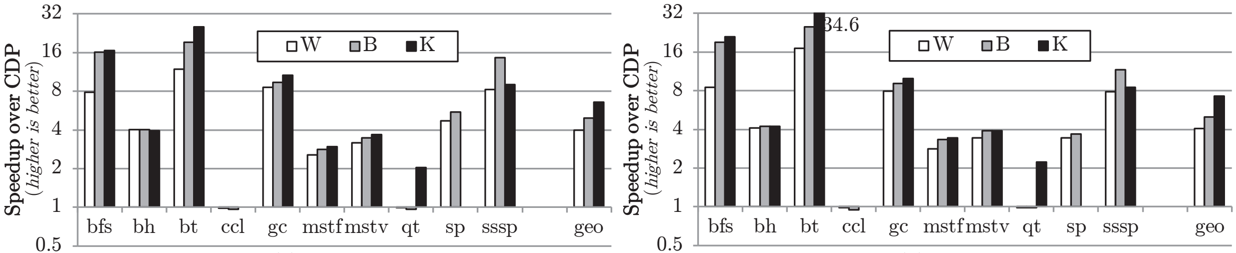
The overall speedup of kernel launch aggregation over CDP is shown in Figure 8 for each benchmark at warp (W), block (B), and kernel (K) granularity. We show results for both Kepler and Maxwell. The similarity of the results demonstrate the portability of our technique across architectures. Because the results are very similar, we only discuss Kepler results in the rest of this section.

All but two benchmarks show speedup over the baseline CDP version for all granularities. All but one show improvement as granularity increases, with geomean speedups of 3.98×, 4.94×, and 6.58× for W, B, and K respectively. Two benchmarks do not have results for kernel-granularity aggregation: *ccl* because it has explicit synchronization and *sp* because the kernel is called in a loop (see Section II-B).

The breakdown of the execution time of each benchmark is shown in Figure 9 for the original CDP version as well as aggregation at each granularity. In the following paragraphs, we discuss each benchmark in the context of these results.

bfs, bt, gc, mstf, mstv, sp. For all six benchmarks, we observe that the original CDP version was dominated by the kernel launch overhead. As the aggregation granularity is increased and fewer kernels are launched, the launch overhead decreases at the expense of additional aggregation logic, resulting in a net performance gain. We also observe a decrease in the amount of time spent doing real work. This is due to the improvement in occupancy. Profiling results show that occupancy for *bfs*, *gc*, *mstf*, *mstv*, and *sp* improves by a geomean of 2.81×, 3.24× and 3.51× for W, B, and K respectively.

sssp. This benchmark behaves very similarly to the previous six, with the difference that performance at kernel granularity



(a) Kepler

(b) Maxwell

Fig. 8. Speedup of Kernel Launch Aggregation

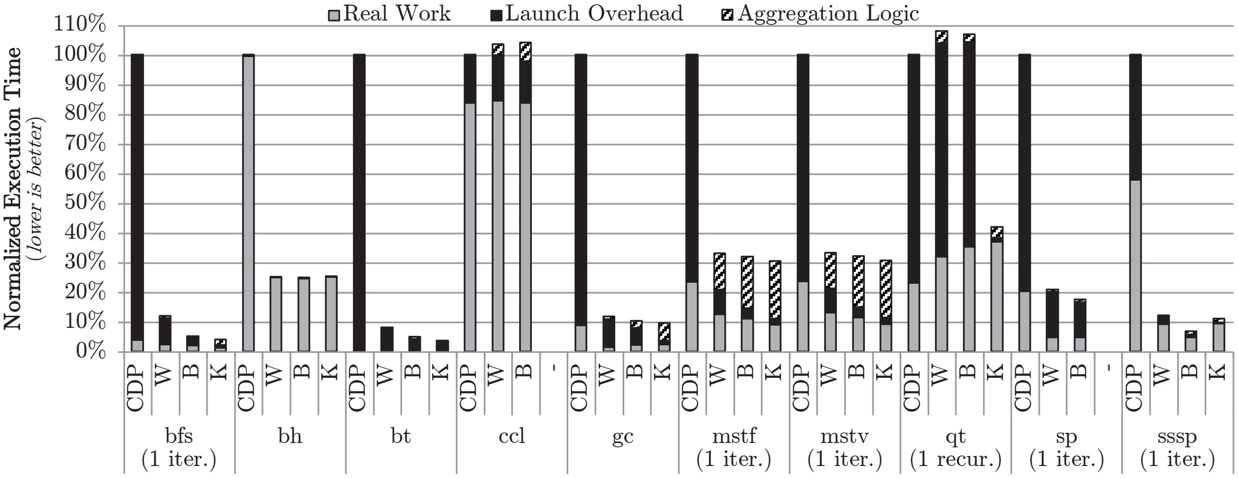


Fig. 9. Breakdown of Execution Time in Kernel Launch Aggregation

aggregation is slightly worse than that at block granularity. That is because block-granularity aggregation finds a better occupancy sweet spot, with occupancy improving by 6.35×, 7.95×, and 6.68× for W, B, and K respectively.

bh. This benchmark originally contains long running children threads so the kernel launch overhead does not dominate performance. For this reason, there is no benefit to be gained from reducing the launch overhead. However, the benchmark still benefits from improved occupancy as with the previous seven. In fact, the occupancy improves by a factor of 3.83×, 3.83× and 3.82× for W, B, and K respectively. These factors are comparable to the factors of improvement in the time spent performing real work shown in the graph.

ccl. This is a unique benchmark because the parent kernel only contains a single thread block with two threads. For this reason, there is not much to be gained from aggregating two launches into one. Instead, some overhead is incurred from the aggregation logic without having much value. However, this benchmark demonstrates that our technique does not significantly harm benchmarks not benefiting from it.

qt. This is a recursive benchmark and also unique because only one thread per block launches a kernel. Therefore warp and block granularity aggregation do not have any impact on the number of kernels launched – they just incur slightly extra overhead. This benchmark again demonstrates that our technique does not harm irrelevant benchmarks. For kernel-granularity aggregation, the launch overhead is significantly reduced.

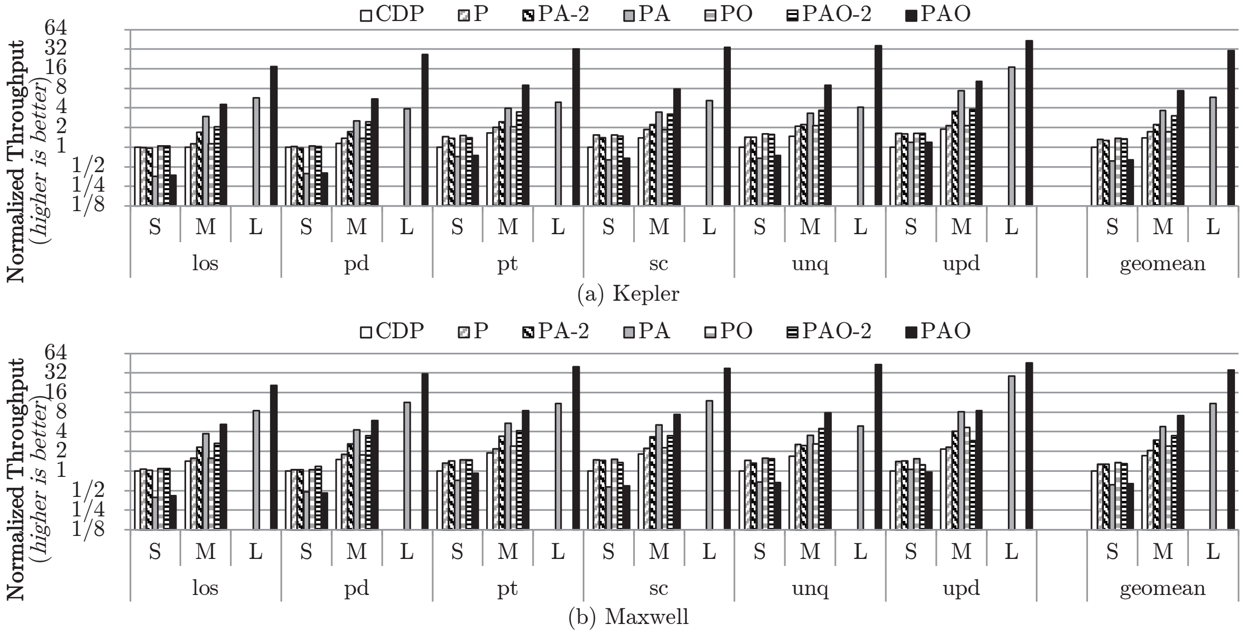
Comparing aggregation with non-CDP, 6 out of the 7 benchmarks having non-CDP versions show speedup from aggregation ranging from 1.2× to 3.7× (geomean 1.6×) over the non-CDP versions. The 7th is *bh* for which the algorithmic changes required to make it amenable to CDP made the naive CDP version significantly slower and hard to recover from. We mention these metrics for quality assurance, but note that the main advantage of CDP over non-CDP is programmability, not performance.

Our results do not show enough variation between granularities to motivate employing a selection technique. However, if needed, one can be employed similar to that in related work [10].

C. Promotion

The overall throughput improvement of kernel launch promotion is shown in Figure 10 for both Kepler and Maxwell. Again, the similarity of the results demonstrate the portability of our technique across architectures, and we only discuss Kepler results in the rest of this section.

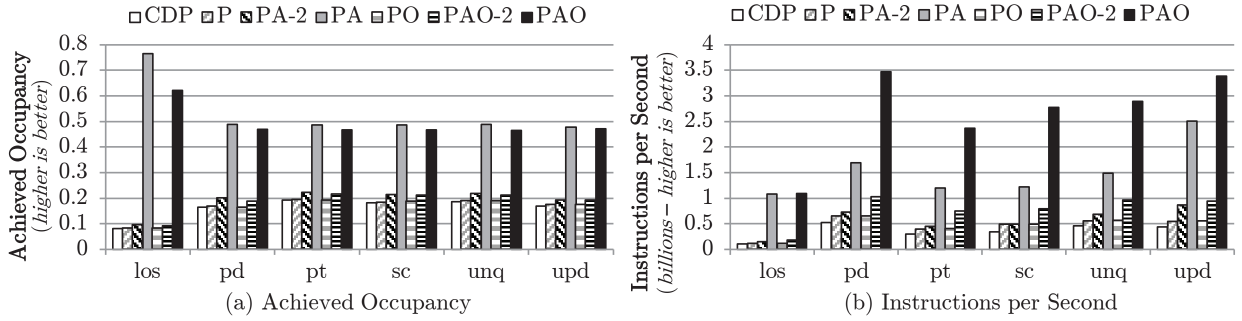
For each benchmark, we show results for the small (S), medium (M), and large (L) datasets. We compare the original CDP version with basic promotion (P), promotion with aggregation where the granularity of aggregation is 2 (PA-2) and



(a) Kepler

(b) Maxwell

Fig. 10. Throughput of Kernel Launch Promotion



(a) Achieved Occupancy

(b) Instructions per Second

Fig. 11. Profile of Kernel Launch Promotion

128 (PA), promotion with overlap (PO), and promotion with aggregation and overlap where the granularity of aggregation is 2 (PAO-2) and 128 (PAO).

Small datasets. The small datasets are designed to create a chain of only two kernel launches. While not realistic, they are intended to show how the overhead of the promotion, aggregation, and overlap logic can cause performance to degrade if the dataset is not big enough. In particular, we point out how the larger aggregation granularity (PA and PAO) results in the most overhead among all when the spawned children are not used. This extreme case demonstrates the tradeoff of aggregation granularity in the overhead it adds to the benchmark to aggregate the kernel launch as well as terminate unused thread blocks.

Medium datasets. The medium datasets are the maximum size that can be executed in CDP, beyond which the call depth limit is no longer sufficient. All benchmarks show speedup at this scale. PAO on the medium dataset has a geomean speedup of 5.27× over CDP on the medium dataset. One interesting observation is that across the benchmarks, PO alone shows little speedup over P, and PA alone shows moderate speedup over P, however PAO shows higher speedup greater than the product of the two. This indicates that aggregation and overlap mutually benefit each other. We show why shortly when we discuss profiling results.

Large datasets. For the large datasets, only PA and PAO can complete while the others are limited by the maximum call depth. This demonstrates the power of our technique in overcoming the call depth limitation through vertical aggregation. PAO on the large datasets achieves a throughput that is 30.44× and 21.81× higher than that achieved by CDP on the small and medium datasets respectively.

Profiling. The profiling results for each of the promotion transformations on the medium datasets are shown in Figure 11. In Figure 11(a), PA and PAO both achieve significantly better occupancy than the other versions. This demonstrates the power of the aggregation optimization in improving the occupancy of dynamic parallelism on the GPU. Despite having the same occupancy, Figure 11(b) shows that PAO executes more instructions per second than PA. This demonstrates the effectiveness of the overlap optimization in making more work available sooner. It is interesting that the overlap optimization does not improve instruction throughout when applied alone, but only when applied with aggregation. That is because without aggregation, PO has low occupancy so it cannot take advantage of available instructions.

V. Related Work

Multiple studies [9], [6], [7] have identified the inefficiency of the current practice of dynamic parallelism, and have proposed different solutions to overcome these inefficiencies.

A. Kernel Launch Aggregation

Hardware-based Aggregation. Dynamic Thread Block Launch (DTBL) [9] performs kernel aggregation in hardware via aggregation tables for buffering kernel calls. This work is further enhanced with a locality-aware scheduler [28]. Orr et al. [29] earlier proposed a similar hardware-based aggregation scheme for fine-grain tasks in HSA processors. Both techniques require hardware modification and are not available on current GPUs, whereas KLAP is a compiler-based approach which performs aggregation on existing devices. Once hardware is available, we expect that hardware- and software-based aggregation can operate synergistically. On the one hand, hardware improvements to dynamic parallelism can improve KLAP's performance further. In return, KLAP can also assist hardware techniques by reducing pressure on the aggregation table/buffer to avoid spilling to memory when there is not enough table space.

An important difference between KLAP and DTBL is that KLAP aggregates kernels before they are launched to reduce the number of launches, whereas DTBL aggregates them after they are launched but makes the launch more lightweight. It is plausible that a KLAP-like approach where kernels are aggregated before launching can also be done in hardware.

Compiler-based Optimization of Dynamic Parallelism. CUDA-NP [7] is a compiler approach that takes advantage of nested parallelism within threads by assigning them additional slave threads in the same thread block to perform their child tasks. This approach, however, is limited by the parallelism available within a single thread block.

The most similar work to our kernel launch aggregation is Free Launch (FL) [10] which performs aggregation-like transformations and eliminates subkernel calls entirely by reusing parent threads to perform child computation. Thus, FL can potentially eliminate more kernel launch overhead and achieve higher speedup than KLAP. The best FL technique relies on using persistent threads [30]. In KLAP, we do not eliminate subkernel calls but rather we demonstrate that CDP can work efficiently after proper aggregation techniques are applied, delivering comparable speedups. Moreover, by using CDP, we avoid the use of persistent threads. This approach has several advantages.

First, KLAP can handle general cases with variable child block sizes and does not need to know the maximum child block size on the host prior to execution. On the other hand, because FL uses persistent threads, the maximum child block size must be known on the host prior to execution in order to ensure that the persistent thread blocks have enough threads to execute the child blocks. Otherwise, the child threads would need to be coarsened to reduce the block size or the maximum possible block size would need to be used, which may not achieve the best occupancy.

Second, KLAP transforms each kernel separately, independent of its caller and callee kernels. This makes our compiler more scalable and our transformations less complex (hence, more robust) as the number of kernels involved in the call hierarchy increases. Our approach already supports multi-depth and recursive call lineages naturally without the need for additional support. Moreover, our approach guarantees memory consistency between parent and child via the semantic of kernel calls, and does not need to place memory fences within arbitrary control flow paths of the parent kernel, which can be both difficult and error-prone.

Third, KLAP does not need to deal with the caveats of persistent threads. For example, kernels with persistent threads must grab all the resources they require for the entirety of their execution, even if they do not need them during some parts of their lifetime. This means that such kernels cannot share their resources with co-runner kernels. On the other hand, KLAP kernels behave like regular kernels and act flexibly in collaborative environments.

We also note that the speedups we report are not directly comparable to those reported by FL for two reasons. The first reason is that we use a faster baseline which uses private streams more efficiently. The second reason is that our benchmarks use a variety of children block sizes, whereas most of FL's benchmarks use large parent block size (1024) and small children block sizes (32). Setting the children block size to 32 enables FL's transformation to execute multiple child blocks concurrently in the same parent block (each parent warp executes a child block). To establish a direct comparison, we reached out to the authors who shared their *bfs* code with us [31]. For the configurations they use, their code is 2.08× faster than ours. For other configurations, the difference is as low as 1.26×. This verifies that CDP can indeed be efficient if the proper compiler transformations are applied.

Other Related Work. Guevara et al. [32] merge a few independent kernels launched from the host together to achieve concurrency. This issue was solved by Fermi's concurrent kernel execution. KLAP merges many identical kernels launched from the device to reduce the number of launches. Li et al. [33] propose a set of parallelization templates that optimize irregular nested loops and parallel recursive computations, particularly for tree and graph algorithms. KLAP applies compiler transformations to convert naive CDP into more efficient aggregated CDP. Ren et al. [34] propose a vector parallelization transformation to "aggregate" similar tasks in recursive task-parallel programs for better utilization of vector hardware. KLAP focuses on aggregation techniques for data-parallel programs in CUDA or OpenCL.

B. Kernel Launch Promotion

Decoupled Software Pipelining [35] extracts parallelism from sequential loops by prioritizing execution of instructions on the (recurrence) critical path. Promotion builds on this idea and extracts parallelism from long dependence chains by promoting (thus prioritizing) launches on the critical path.

Producer-consumer patterns on GPUs have received moderate attention. Kernel Weaver [36] fuses producer-consumer kernels performing relational algebra operations, but this was before dynamic parallelism was introduced. KLAP specifically targets producer-consumer patterns expressed with dynamic parallelism. Our benchmarks include relational operations.

Gómez-Luna et al. [5] introduce a set of GPU algorithms (called data sliding algorithms) that perform promotion with aggregation and overlap through libraries. However, Gómez-Luna et al. do not connect promotion techniques with recursion using dynamic parallelism for better programmability. KLAP makes this connection, and introduces the promotion techniques to resolve the depth limitation of dynamic parallelism. It applies compiler transformations to convert naive CDP to promoted CDP. KLAP further isolates the effects of promotion, aggregation, and overlap.

VI. Conclusion

In this paper, we have presented KLAP, a set of compiler techniques that improve the performance of programs that use dynamic parallelism on current hardware. KLAP is comprised of two main transformation techniques: kernel launch aggregation and kernel launch promotion.

Kernel launch aggregation fuses kernels launched by multiple threads into a single aggregated kernel launch. It can be applied at warp, block, or kernel granularity. The benefit of aggregation is that it reduces the number of launches and makes kernels coarser which improves device occupancy.

Kernel launch promotion optimizes kernels with producer-consumer relations by launching children kernels earlier than their original call site and enforcing dependence via release-acquire chains. Promotion also enables two further optimizations: aggregation and overlap. Aggregation vertically fuses descendants into a single aggregated kernel which improves occupancy and addresses the problem of limited call depth. Overlap increases instruction throughput by executing the independent part of a child kernel concurrently with its parent.

We have shown that kernel launch aggregation at varying granularities and kernel launch promotion with aggregation and overlap result in significant speedups on multiple architectures for a variety of applications using dynamic parallelism.

Acknowledgments

This work is supported by the NVIDIA GPU Center of Excellence at the University of Illinois, Hewlett-Packard Enterprise Labs, the Starnet Center for Future Architecture Research (C-FAR), and the Blue Waters PAID Use of Accelerators (NSF OCI 07-25070 490595).

```
__global__ void child(float* param1, unsigned int param2) {
  doChildWork(blockIdx.x, blockDim.x);
}
__global__ void parent(float* param1, unsigned int param2) {
  ...
  if(cond) {
    foo();
    child <<< gD, bD >>> (arg1, arg2);
  }
  ...
}
```

Original Code

```
__global__ void child_agg(float ** __param1_array, unsigned int* __param2_array,
  unsigned int* __gD_array, unsigned int* __bD_array, char* __mem_pool,
  unsigned int* __free_idx) {
  // Identify parent
  unsigned int __parent_id = __block_find_parent_id(__gD_array, blockIdx.x);
  // Load params/configs
  float * param1 = __param1_array[__parent_id];
  unsigned int param2 = __param2_array[__parent_id];
  unsigned int __blockDim_x = __bD_array[__parent_id];
  unsigned int __blockIdx_x =
    blockIdx.x - ((__parent_id == 0)?0:__gD_array[__parent_id - 1]);
  // Execute original kernel code
  if(threadIdx.x < __blockDim_x) {
    doChildWork(__blockIdx_x, __blockDim_x);
} }
__global__ void parent(float* param1, unsigned int param2, char* __mem_pool,
  unsigned int* __free_idx) {
  ...
  unsigned int __pred0 = (cond)?1:0;
  if(__pred0) {
    foo();
  }
  // Setup aggregated kernel launch
  {  // Allocate memory for param/config arrays
    unsigned int __i = threadIdx.x;
    __shared__ char* __local_mem_pool;
    if(threadIdx.x == 0) {
      unsigned int __local_mem_pool_size = blockDim.x*(sizeof(float*)
        + sizeof(unsigned int) + 2*sizeof(unsigned int) /*gD,bD*/);
      __local_mem_pool =
        __mem_pool + atomicAdd(__free_idx, __local_mem_pool_size);
    }
    __syncthreads();
    char* __my_pool = __local_mem_pool;
    float ** __param1_array = (float **) __my_pool;
    __my_pool += blockDim.x*sizeof(float *);
    unsigned int* __param2_array = (unsigned int*) __my_pool;
    __my_pool += blockDim.x*sizeof(unsigned int);
    unsigned int* __gD_array = (unsigned int*) __my_pool;
    __my_pool += blockDim.x*sizeof(unsigned int);
    unsigned int* __bD_array = (unsigned int*) __my_pool;
    // Store params/configs
    __param1_array[__i] = arg1;
    __param2_array[__i] = arg2;
    __gD_array[__i] = __pred0 * gD;
    __bD_array[__i] = bD;
    // Calculate aggregated configs
    unsigned int __sum_gD = __block_inclusive_scan(__gD_array);
    unsigned int __max_bD = __block_max(__bD_array);
    // Launch aggregated kernel
    if(threadIdx.x == blockDim.x - 1) {
      if(__sum_gD > 0) {
        child_agg <<< __sum_gD, __max_bD >>> (__param1_array,
          __param2_array, __gD_array, __bD_array, __mem_pool,
          __free_idx);
} } } }
```

Transformed Code (block-granularity aggregation)

Fig. 12. Detailed Code for Block-granularity Aggregation

References

[1] S. Jones, “Introduction to dynamic parallelism,” in *GPU Technology Conference Presentation*, 2012.

[2] W. H. Wen-mei, *Heterogeneous System Architecture: A new compute platform infrastructure*. Morgan Kaufmann, 2015.

[3] P. Zhang, E. Holk, J. Matty, S. Misurda, M. Zalewski, J. Chu, S. McMillan, and A. Lumsdaine, “Dynamic parallelism for simple and efficient gpu graph algorithms,” in *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*, IA3 ’15, pp. 11:1–11:4, ACM, 2015.

[4] M. L. Sætra, A. R. Brodtkorb, and K.-A. Lie, “Efficient gpu-implementation of adaptive mesh refinement for the shallow-water equations,” *Journal of Scientific Computing*, vol. 63, no. 1, pp. 23–48, 2015.

[5] J. Gómez-Luna, L.-W. Chang, I.-J. Sung, W.-M. Hwu, and N. Guil, “In-place data sliding algorithms for many-core architectures,” in *Parallel Processing (ICPP), 2015 44th International Conference on*, pp. 210–219, IEEE, 2015.

[6] J. Wang and S. Yalamanchili, “Characterization and analysis of dynamic parallelism in unstructured GPU applications,” in *Workload Characterization (IISWC), 2014 IEEE International Symposium on*, pp. 51–60, IEEE, 2014.

[7] Y. Yang and H. Zhou, “CUDA-NP: Realizing nested thread-level parallelism in GPGPU applications,” in *ACM SIGPLAN Notices*, vol. 49, pp. 93–106, ACM, 2014.

[8] “A CUDA dynamic parallelism case study: PANDA.” https://devblogs.nvidia.com/parallelforall/a-cuda-dynamic-parallelism-case-study-panda/. Accessed: 2016-04-01.

[9] J. Wang, N. Rubin, A. Sidelnik, and S. Yalamanchili, “Dynamic thread block launch: A lightweight execution mechanism to support irregular applications on GPUs,” in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pp. 528–540, ACM, 2015.

[10] G. Chen and X. Shen, “Free launch: optimizing GPU dynamic kernel launches through thread reuse,” in *Proceedings of the 48th International Symposium on Microarchitecture*, pp. 407–419, ACM, 2015.

[11] T. Kaldewey, J. Hagen, A. Di Blas, and E. Sedlar, “Parallel search on video cards,” in *Proceedings of the First USENIX Conference on Hot Topics in Parallelism*, HotPar’09, p. 9, USENIX Association, 2009.

[12] D. Merrill, “CUB:kernel-level software reuse and library design,” in *GPU Technology Conference Presentation*, 2013.

[13] P. Jääskeläinen, C. S. de La Lama, E. Schnetter, K. Raiskila, J. Takala, and H. Berg, “pocl: A performance-portable OpenCL implementation,” *International Journal of Parallel Programming*, vol. 43, no. 5, pp. 752–785, 2015.

[14] K. Asanovic, S. W. Keckler, Y. Lee, R. Krashinsky, and V. Grover, “Convergence and scalarization for data-parallel architectures,” in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 1–11, IEEE Computer Society, 2013.

[15] M. Billeter, O. Olsson, and U. Assarsson, “Efficient stream compaction on wide SIMD many-core architectures,” in *Proceedings of the Conference on High Performance Graphics 2009*, HPG ’09, pp. 159–166, ACM, 2009.

[16] L. Howes and A. Munshi, “The OpenCL specification, version 2.0,” *Khronos Group*, 2015.

[17] S. Yan, G. Long, and Y. Zhang, “StreamScan: Fast scan algorithms for GPUs without global barrier synchronization,” in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP ’13, pp. 229–238, ACM, 2013.

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

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

[20] NVIDIA, “CUDA samples v. 7.5,” 2015.

[21] Y. Ukidave, F. N. Paravecino, L. Yu, C. Kalra, A. Momeni, Z. Chen, N. Materise, B. Daley, P. Mistry, and D. Kaeli, “NUPAR: A benchmark suite for modern GPU architectures,” in *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, ICPE ’15, pp. 253–264, ACM, 2015.

[22] A. V. P. Grosset, P. Zhu, S. Liu, S. Venkatasubramanian, and M. Hall, “Evaluating graph coloring on GPUs,” in *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP ’11, pp. 297–298, ACM, 2011.

[23] “Matrix market.” http://math.nist.gov/MatrixMarket/. Accessed: 2016-04-01.

[24] N. Bell and J. Hoberock, “Thrust: a productivity-oriented library for cuda,” *GPU Computing Gems: Jade Edition*, 2012.

[25] “GPU pro tip: CUDA 7 streams simplify concurrency.” http://devblogs.nvidia.com/parallelforall/gpu-pro-tip-cuda-7-streams-simplify-concurrency/. Accessed: 2016-04-01.

[26] “CUDA dynamic parallelism API and principles.” https://devblogs.nvidia.com/parallelforall/cuda-dynamic-parallelism-api-principles/. Accessed: 2016-04-10.

[27] NVIDIA, “Profiler user’s guide v. 7.5,” 2015.

[28] J. Wang, N. Rubin, A. Sidelnik, and S. Yalamanchili, “Laperm: Locality aware scheduler for dynamic parallelism on gpus,” in *The 43rd International Symposium on Computer Architecture (ISCA)*, June 2016.

[29] M. S. Orr, B. M. Beckmann, S. K. Reinhardt, and D. A. Wood, “Fine-grain task aggregation and coordination on GPUs,” in *ACM SIGARCH Computer Architecture News*, vol. 42, pp. 181–192, IEEE Press, 2014.

[30] K. Gupta, J. A. Stuart, and J. D. Owens, “A study of persistent threads style GPU programming for gpgpu workloads,” in *Innovative Parallel Computing (InPar), 2012*, pp. 1–14, IEEE, 2012.

[31] G. Chen and X. Shen. private communication.

[32] M. Guevara, C. Gregg, K. Hazelwood, and K. Skadron, “Enabling task parallelism in the cuda scheduler,” in *Workshop on Programming Models for Emerging Architectures*, vol. 9, 2009.

[33] D. Li, H. Wu, and M. Becchi, “Nested parallelism on GPU: Exploring parallelization templates for irregular loops and recursive computations,” in *Parallel Processing (ICPP), 2015 44th International Conference on*, pp. 979–988, IEEE, 2015.

[34] B. Ren, Y. Jo, S. Krishnamoorthy, K. Agrawal, and M. Kulkarni, “Efficient execution of recursive programs on commodity vector hardware,” in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 509–520, ACM, 2015.

[35] R. Rangan, N. Vachharajani, M. Vachharajani, and D. I. August, “Decoupled software pipelining with the synchronization array,” in *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pp. 177–188, IEEE Computer Society, 2004.

[36] H. Wu, G. Diamos, S. Cadambi, and S. Yalamanchili, “Kernel weaver: Automatically fusing database primitives for efficient GPU computation,” in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 107–118, IEEE Computer Society, 2012.