Free Launch: Optimizing GPU Dynamic Kernel Launches through Thread Reuse

Guoyang Chen, Xipeng Shen
Computer Science Department, North Carolina State University
890 Oval Drive, Raleigh, NC, USA 27695
gchen11@ncsu.edu, xshen5@ncsu.edu

ABSTRACT

Supporting dynamic parallelism is important for GPU to benefit a broad range of applications. There are currently two fundamental ways for programs to exploit dynamic parallelism on GPU: a software-based approach with software-managed worklists, and a hardware-based approach through dynamic subkernel launches. Neither is satisfactory. The former is complicated to program and is often subject to some load imbalance; the latter suffers large runtime overhead.

In this work, we propose *free launch*, a new software approach to overcoming the shortcomings of both methods. It allows programmers to use subkernel launches to express dynamic parallelism. It employs a novel compiler-based code transformation named *subkernel launch removal* to replace the subkernel launches with the reuse of parent threads. Coupled with an adaptive task assignment mechanism, the transformation reassigns the tasks in the subkernels to the parent threads with a good load balance. The technique requires no hardware extensions, immediately deployable on existing GPUs. It keeps the programming convenience of the subkernel launch-based approach while avoiding its large runtime overhead. Meanwhile, its superior load balancing makes it outperform manual worklist-based techniques by 3X on average.

Categories and Subject Descriptors

3.4 [**Programming Languages**]: Processors—*optimization, compilers*

Keywords

GPU, Dynamic Parallelism, Optimization, Thread Reuse, Compiler, Runtime Adaptation

MICRO 2015 Waikiki, Hawaii USA
Copyright 2015 ACM ISBN 978-1-4503-4034-2/15/12 ...$15.00.
DOI: http://dx.doi.org/10.1145/2830772.2830818.

1. INTRODUCTION

As GPU shows some great potential for accelerating general-purpose computations, many emerging data intensive applications are trying to exploit GPU for accelerations. These applications range from business analytics, user modelling, to social network analysis and so on. They center around sophisticated algorithms from Machine Learning, Graph Theory, Deductive Reasoning and others. These algorithms often feature irregularities and dynamic parallelism.

Breadth-first search (BFS), for example, needs to traverse all nodes in a graph in a breadth-first order. When it reaches a node, it can process all its neighbors in parallel. The amount of parallelism is determined by the degree of the node, which often differs from one node to another.

There have been two fundamental ways to exploit dynamic parallelism on GPU. (1) The first is software-based techniques with software-managed worklists. They use some carefully designed data structures (e.g., double buffers [1]) to store parallel tasks for GPU threads to grab and execute. (2) The second is hardware-based techniques, represented by the CUDA Dynamic Parallelism (CDP) [2] that NVIDIA introduces to their recent GPU and the corresponding technique that OpenCL [3] introduces lately. It is also called device-side nested kernel launch capability or *subkernel launch (SKL)* in short. This feature allows a GPU kernel to launch a child kernel at runtime through some programming interface. It is more intuitive and simpler to use than the software approach. For the aforementioned BFS example, one may simply launch a subkernel to process the neighbors of a node.

Neither solution however is satisfactory. The software-based techniques are complicated to program and maintain, and even sophisticated implementations still have considerable load imbalance among threads, leaving substantial room for performance improvement (as to be shown in Section 5). The hardware-based techniques are subject to large time overhead of subkernel launches. The overhead consists of the time to save the states (registers and shared memory content) of the parent threads, to create the child kernel and threads, and to restore the states of the parent threads. For the nature of massive parallelism of GPU, each of the three components could take some substantial amount of time, causing overhead sometimes even greater than the workload itself [4, 5, 6]. So despite being intuitive to use, this method has received only little practical usage. Some recent hardware extensions have been proposed to help alleviate the problem [5, 7]. They are yet to be adopted in the actual GPU designs.

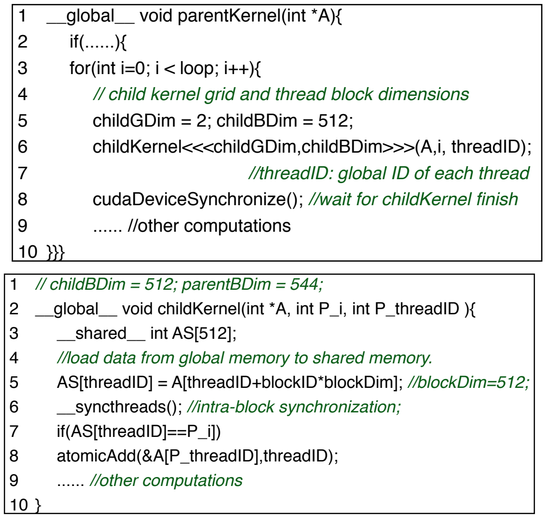
In this work, we propose a new software solution named *free launch*, which replaces subkernel launches with parent thread reuse. It starts with a program that uses subkernel launch for dynamic parallelism. It then automatically transforms the program to remove those subkernel launches, and assigns the work of the subkernels to the parent GPU threads in a balanced manner. By allowing the use of subkernel launches in the original program, *free launch* keeps the conveniences of the hardware-based approach for programmers. By automatically removing the subkernel launches, it avoids the large runtime overhead of the hardware-based approach. Meanwhile, its adaptive assignment of tasks to parent threads gives it the opportunity to achieve a better load balance and hence outperform the previous software-based manual techniques. *Free launch* requires no hardware extensions, immediately deployable on current applications and GPUs.

The main challenge for developing *free launch* is how to materialize its idea in a sound and effective manner, such that, after the removal, the program can run correctly and efficiently. We propose two techniques to achieve the goal. The first is *subkernel launch removal* (or *launch removal* in short), which is a new compiler-based code transformation for replacing subkernel launches with parent thread reuses. The second is *adaptive subtask assignment* (a **subtask** in this paper refers to the set of work a child thread block is supposed to process), which, based on our exploration of four load-balancing schemes, equips the *launch removal* with the capability to produce subtask-to-thread assignments with a superior load balance.

Experiments on a set of kernels and two GPUs show that *free launch* can effectively support dynamic parallelism while avoiding the shortcomings of prior software and hardware techniques. It outperforms the manual worklist-based method by 3X on average, thanks to the better load balance brought by its adaptive subtask assignment. Compared to versions that completely rely on existing hardware support of subkernel launches for dynamic parallelism, it gives 36X average speedups for its removal of the large launching overhead and the better load balancing.

Overall, this work makes several major contributions:

- It proposes a novel software method, *free launch*, to support dynamic parallelism on GPU; the method overcomes the main drawbacks of prior software and hardware techniques.
- It develops a new code transformation, *launch removal*, that is able to automatically replace subkernel launches with reuses of parent threads.
- It develops *adaptive subtask assignment*, an approach to adaptively selecting the load-balancing subtask assignment scheme that suites a given kernel.
- It shows that *free launch* offers a significantly better support to dynamic parallelism on existing GPU, with 3X average speedups over manual worklist-based methods and 36X average speedups over subkernel launch-based methods.



```
__global__ void parentKernel(int *A){
   if(......){
   for(int i=0; i < loop; i++){
      // child kernel grid and thread block dimensions
      childGDim = 2; childBDim = 512;
      childKernel<<<childGDim,childBDim>>>(A,i, threadID);
                              //threadID: global ID of each thread
      cudaDeviceSynchronize(); //wait for childKernel finish
      ...... //other computations
}}}
```

```
// childBDim = 512; parentBDim = 544;
__global__ void childKernel(int *A, int P_i, int P_threadID ){
   __shared__ int AS[512];
   //load data from global memory to shared memory.
   AS[threadID] = A[threadID+blockID*blockDim]; //blockDim=512;
   __syncthreads(); //intra-block synchronization;
   if(AS[threadID]==P_i])
   atomicAdd(&A[P_threadID],threadID);
   ...... //other computations
}
```

Figure 1: DynCall: An example codelet representing GPU kernels with dynamic parallelism.

2. LAUNCH REMOVAL TRANSFORMATION

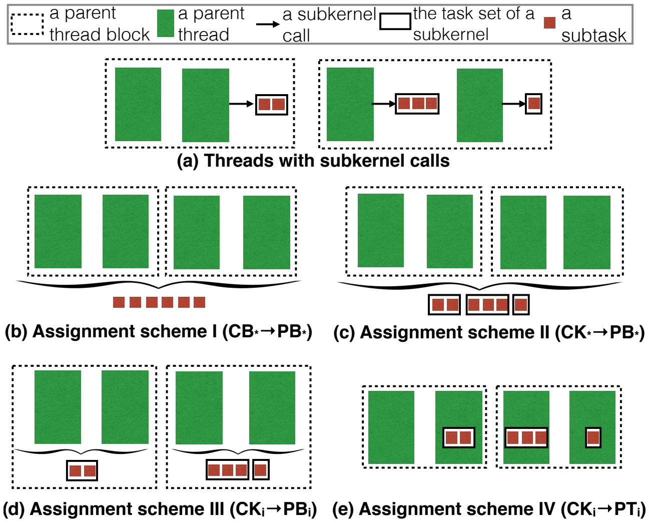
Launch removal centers on reusing parent threads rather than creating children threads to process subtasks. It enables the reuse through four types of compiler-based program transformations. This section presents these transformations.

2.1 A Running Example

To help explain our proposed transformations, we first introduce a running example as shown in Figure 1. The codelet, which we call *DynCall*, represents a general form of a GPU kernel with children kernel calls. The loop surrounding the child kernel call represents the case in which a parent thread may invoke the child kernel repeatedly. In many GPU programs, only a portion rather than all of the parent threads launch a child kernel. The *if* statement in the *DynCall* codelet represents the condition that a parent thread needs to meet such that it needs to launch a child kernel. We call it the *launching condition*. It affects the number of children kernels to be launched by the parent kernel and hence the collection of subtasks to be generated, which in turn affects the appropriate assignment of subtasks to parent threads. To handle the variations, we design four types of subtask assignments as explained next.

2.2 Four Types of Subtask Assignments

There are many possible variations in the amount and patterns of the dynamic parallelism associated with a



(a) Threads with subkernel calls

(b) Assignment scheme I ($CB_* \to PB_*$)

(c) Assignment scheme II ($CK_* \to PB_*$)

(d) Assignment scheme III ($CK_i \to PB_i$)

(e) Assignment scheme IV ($CK_i \to PT_i$)

Figure 2: Illustration of the four assignments (b,c,d,e) of subtasks enabled by the *launch removal* program transformations on a kernel with subkernel calls (a).

child kernel. For instance, depending on the launching condition, in some instances of *DynCall*, every parent thread launches a child kernel; in some other instances, only some do. Moreover, the children kernels launched by different parent threads could have different numbers of thread blocks and dimensions. As a result, the suitable assignments of subtasks to parent threads vary accordingly.

By examining some kernels with dynamic parallelism and their manually developed more efficient counterparts without using dynamic parallelism, we find that four types of subtask assignments offer different tradeoffs in load balance and control overhead, and together can meet the needs of most kernels for effective subtask assignments. Figure 2 illustrates these different schemes. Figure 2 (a) shows a very simple kernel that uses subkernel launches. The kernel consists of two thread blocks with each containing only two threads. The second thread in the first thread block has a subkernel launch, which is used to process two subtasks; the two threads in the second thread block each have a subkernel launch with three and one subtask respectively. Figures 2 (b,c,d,e) illustrate the four types of subtask assignments respectively. We next explain each of them (some notations: C for child, P for parent, K for kernel, B for thread block, T for thread, → for assignment.)

- $CB_* \to PB_*$: This subtask assignment is at the thread block level. The assignment unit is a subtask. The subscript "*" means that any subtask is possible to be assigned to any parent thread block. As Figure 2 (b) shows, any of the five subtasks that the subkernels in Figure 2 (a) are to process, could get assigned to any of the two thread blocks. This global assignment could incur some runtime control overhead, but it is, among these four types of assignments, the most general way to ensure a good load balance among parent threads.

- $CK_* \to PB_*$: This assignment scheme works at the level of a subkernel. The assignment unit is the whole set of subtasks of a subkernel. As Figure 2 illustrates, there are three sets of subtasks shown as the three solid-lined boxes, corresponding to the three subkernel launches shown in Figure 2 (a). Each of the three sets could be assigned to any of the thread blocks. Because a task set is the assignment unit, unlike in the previous scheme, the tasks inside a set cannot get assigned to different thread blocks. This assignment could help ensure a good load balance and at the same time avoid some control overhead needed for handling the tasks of different children blocks.

- $CK_i \to PB_i$: This assignment is similar to the previous one, except that the assignment is local: The i^{th} parent block handles the subtasks of only the child kernel that is supposed to be launched by itself. As illustrated by Figure 2 (d), the first thread block can handle only the first set of subtasks as that is the only set of subtasks that the children threads of the first thread block are supposed to process. For the same reason, the second thread block handle the other two sets of subtasks. This local assignment is not as flexible as the aforementioned ones in load balancing, but is simple and avoids much runtime overhead.

- $CK_i \to PT_i$: This assignment is similar to the previous scheme except that it assigns tasks to a parent thread rather than a parent thread block. In this scheme, a parent thread needs to process the set of subtasks that its children threads are originally supposed to process. As illustrated by Figure 2 (e), the second thread will handle the first set of subtasks, the third thread will handle the second set, and the fourth thread will handle the third set. This assignment is not as flexible, but is the simplest to implement and may still suit some scenarios.

These four types of assignments feature different strengths. We postpone to Section 3 the discussion on how to select them effectively. This section describes how code transformations can materialize each of them.

2.3 Transformation for $CB_* \to PB_*$

The transformation for $CB_* \to PB_*$ involves almost all the intricate aspects of launch removal transformations. Understanding this transformation would make the other transformations easy to understand. Figure 3 shows the transformed result of our example DynCall. We explain the transformation in three steps.

```
1        __device__ int kernelCalls = 0, ChildTBs = 0; //Global variables.
2        __global__ void parentKernel(int *A,record *worklist, int BLOCK, int THREAD){
3        //worklist: record all launching info; BLOCK: # of PTBs in original kernel;
4        //THREAD: # of threads in original PTB.
5        int i; int i_p=0; bool hasLaunch = False;     // initialization.
6        //assign original parent TBs to persistent TBs; gridDim: # of persistent TBs.
7        for(i_p=0; i_p+blockID<BLOCK && threadID<THREAD; i_p+=gridDim){
8            if(……){
9                childGDim = 2; childBDim = 512;
10               for(i=0; i < loop; i++){
11                   int curNumCalls = atomicAdd(&kernelCalls, 1);
12                   hasLaunch = True;
13                   atomicAdd(&ChildTBs, childGDim);
14                   worklist[curNumCalls] = {childGDim, childBDim, i, threadID};
15                   goto P;   //childKernel<<<grid,block>>>(i, threadID);
16   Back:           ......        //other computations
17       }}}
18       // Code for  subtask processing
19   P:  syncAllThreads(); //synchronize all thread blocks (via usage of persistent threads)
20        if(kernelCalls>0){ //assign and process subtasks.
21           int tasksPerPTB =(ChildTBs + gridDim-1)/gridDim;
22           for(int i_c=0; i_c<tasksPerPTB && tasksPerPTB *blockID+i_c<ChildTBs; i_c++){
23               retrieve {childGDim, childBDim, P_i,P_threadID} from worklist;
24               int NewBlockID = find_blockID_for_launch(tasksPerPTB *blockID+i_c);
25               code of the work of a child thread;
26           }
27           syncAllThreads();
28           ...... //reset kernelCalls and ChildTBs to 0.
29           syncAllThreads();
30           if(hasLaunch) {hasLaunch = False; goto Back;}
31           else goto P;
32       }}
```

Figure 3: Transformation result for $CB_* \rightarrow PB_*$. The black code is what the transformation inserts; other code is from the original program.

Making Parent Threads Persistent.

Due to the global nature of the task assignments in this transformation, it is necessary to have synchronizations across thread blocks. GPU, by default, supports intra-block but not inter-block synchronizations. Inter-block synchronization is possible only when the GPU kernel code uses persistent threads. So, the first step in our code transformation is to make parent threads persistent.

The idea of persistent threads is first described in a paper by Gupta and others [8]. The main idea is to keep threads resident on SMs until kernel finishes. During the kernel execution, every persistent thread block continuously fetches some other task from the remaining unfinished task queue when it finishes a task (a task refers to the work an original thread block is supposed to do).

In Figure 3, the "for" loop at line 7 realizes a mapping from the original parent thread blocks to the persistent blocks. In each iteration of the loop executed by a persistent thread block, the work of the original parent thread block (whose ID equals i_p) is processed by that persistent thread block.

Our transformation ensures that the total number of persistent thread blocks is no larger than the maximum number of thread blocks that can concurrently run on a GPU (by considering the shared memory usage and register usage of the kernel). This condition makes synchronizations across all thread blocks feasible. The cross-block barrier "syncAllThreads" in Figure 3 (lines 19, 27, 29) is implemented as follows: It tries to synchronize threads in the current thread block first and then repeatedly checks a counter on the global memory to wait for all thread blocks to reach here.

Parameters and Shared Memory.

A child kernel launch involves some parameters, including the launch configuration (grid and block dimensions), and the function arguments. These parameters determine both the number of subtasks and the operations in the subtasks. For parent threads to fulfill these subtasks, these parameters must be recorded. Line 14 in Figure 3 does it, where "curNumCalls" is the index number of the child kernel call under consideration.

The usage of shared memory in GPU adds some special complexities to the transformation. Shared memory in GPU is some on-chip memory that can be shared by threads in the same thread block. A good usage of it is essential for performance for its low latency. It is limited in size (48KB per SM on NVIDIA GPU), but could be needed by the parent threads tasks and the subtasks. We summarize the intricacies in handling shared memory in our code transformation in various scenarios as follows:

Scenario 1. The child kernel launch is asynchronous. In this case, our code generation omits the "goto" statement in line 15 in Figure 3 such that all parent threads will finish their own work first and then go to line 19 to do the subtasks. Shared memory used in the work of the parent threads is reused for the subtask processing. The launch of the parent kernel is changed such that the size of shared memory for a thread block is set as MAX{shared_size_parent, shared_size_child} to accommodate the needs.

Scenario 2. The total shared memory usage of parent kernel and child kernel doesn't affect the maximum number of thread blocks that can run concurrently. That means, the maximum occupancy of the GPU is determined by some other resource (e.g., registers) rather than shared memory. In this situation, a parent thread block uses one shared memory array to do its own work and use another shared memory array for the subtasks.

Scenario 3. In other cases, the transformed code makes the parent threads first save its shared memory content to global memory before using shared memory to do subtasks. It is facilitated with the global barrier *syncAllThreads()*. When subtasks are finished, the content of the shared memory gets restored before the parent threads resume its own work. Similar to scenario 1, the shared memory array size is set to MAX{shared_size_parent, shared_size_child}.

Control Flows and Subtask Assignment.

Control flow changes are an important part of the transformation. They are materialized by the several "goto" statements in Figure 3. The first "goto" (line 15) corresponds to a scenario where the child kernel launch is a synchronous launch. The "goto" makes the par-

```
22        for(int i_c=0; i_c+blockID<kernelCalls; i_c+=gridDim){
23            retrieve {childGDim, childBDim, P_i,P_threadID} from worklist;
24            for(int i_cb=0; i_cb<childGDim && threadID<childBDim; i_cb++){
25                code of the work of a child thread; //with blockID replaced by i_cb.
26            }}
```

Figure 4: The main part of the $CK_* \rightarrow PB_*$ transformation that differs from the $CB_* \rightarrow PB_*$.

ent threads jump to subtasks before continuing its own work. The “syncAllThreads” at line 19 ensures that all threads synchronized before starting processing subtasks to avoid data races on processing the set of subtasks. The “for” loop at line 22 evenly partitions the subtasks among parent thread blocks; “tasksPerPTB” is the number of subtasks per parent thread block (PTB). In the i_c'th iteration of the loop, the parent thread block whose ID is “blockID” processes the (tasksPerPTB *blockID+i_c)'th subtask. Before the actual processing, launching parameters are retrieved from “worklist” (line 23).

After all subworks have been finished, *kernelCalls* and *ChildTBs* get reset. Parent threads that have left their own work to process subtasks (their local variable “hasLaunch” is “true”) go back to do their own work and continue their executions (the “goto” at line 30). Note that some threads may not have met the “if” condition at line 8, which means that they were not supposed to launch children kernels. In the transformed code, these threads arrive at the synchronization point at Line 19 with “hasLaunch” equalling “false”. After they help process some subtasks, through the “goto” at line 31, they return to Line 19 as they have finished all of their own work. They continue to wait to process some future subtasks that their peers may launch.

2.4 Transformation for $CK_* \rightarrow PB_*$

This transformation tries to assign a parent thread block with the subtasks of an entire child kernel launch rather than of a child thread block. The resulting code is similar to what Figure 3 shows. The main difference is that lines 22 to 26 in Figure 3 are replaced with the code in Figure 4. In each iteration of the outer loop in Figure 4, the sets of subtasks of a child kernel call are processed by a parent thread block. In each iteration of the inner loop, the parent thread block processes one subtask. Another difference is that since the new assignment does not need “tasksPerTB” anymore, statements relevant to it are removed (lines 13 and 21).

2.5 Transformation for $CK_i \rightarrow PB_i$

This transformation tries to reuse all threads in a parent thread block to do the subtasks of the children kernels that the parent thread block itself launches. The generated code is similar to Figure 3; there are two main differences. First, the assignment loop (lines 22 to 26) in Figure 3 is replaced with the code in Figure 5 to materialize the different subtask assignment. Variables *wordlist* and *kernelCalls* become local to a parent thread block, and *kernelCalls* is put onto shared memory for efficiency. Second, because the assignment is local to a thread block, there is no need for inter-block synchronizations and hence no need for persistent threads anymore: The “for” loop on line 7 in Figure 3 is replaced with a simple check “if(threadID<THREAD)” just to ensure that the current thread is within a valid range. The “synAllThreads” calls on lines 19, 27, 29 are replaced with “__synchthreads” for intra-block synchronizations. Similar to $CK_* \rightarrow PB_*$, statements relevant to “tasksPerTB” (lines 13 and 21) are removed.

```
22      for(int i_c=0; i_c<kernelCalls; i_c++){
23          retrieve {childGDim, childBDim, P_i,P_threadID} from worklist;
24          for(int i_cb=0; i_cb<childGDim && threadID<childBDim; i_cb++){
25              code of the work of a child thread;  // blockID replaced with i_cb.
26      }}
```

Figure 5: The main part of the $CK_i \rightarrow PB_i$ transformation that differs from the $CB_* \rightarrow PB_*$. The *worklist* becomes a variable local to a parent thread block.

```
1   __global__ void parentKernel(int *A){
2       if(......){
3           childGDim = 2; childBDim = 512;
4           for(int i=0; i < loop; i++){
5               //childKernel<<<childGDim,childBDim>>>(A, i, threadID);
6               one thread does all subtasks of a child kernel.
7               ...... //other computations
8   }}}
```

Figure 6: Transformation result for $CK_i \rightarrow PT_i$.

2.6 Transformation for $CK_i \rightarrow PT_i$

Figure 6 shows the transformation of $CK_i \rightarrow PT_i$. The locally-scoped nature makes this the simplest among all the transformations. Line 6 in Figure 6 ensures that only the thread that is supposed to launch a child kernel will do all subtasks of that child kernel.

2.7 Other Complexity

There is some subtlety when the thread block size of the child kernel is smaller than the parent kernel. Consider a case where a child thread block has 512 threads while a parent thread block has 544 threads. The original child kernel has a “__synchthreads” to synchronize all the 512 threads in a child thread block. So in the transformed code, when the parent threads reach the synchronization statement in the subtask, we only want its first 512 threads to synchronize to emulate the operations of the child threads. However, the default “__synchthreads” statement would synchronize all 544 threads. We use CUDA customized synchronization to solve the problem. Customized synchronization is some PTX assembly instruction, which allows partial thread block sync. The PTX code is ”bar.sync a{,b}”, where a is the ID of the partial barrier, and b specifies the number of threads participating in the barrier.

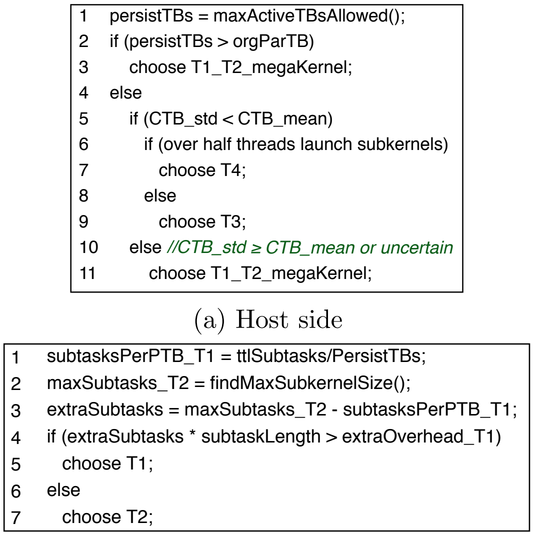
3. TASK ASSIGNMENT SELECTION

The four transformations have different strengths in load balance and runtime control overhead. From T1 to T4, the flexibility in load balancing gets weaker, but at the same time, the needed runtime control becomes simpler and less costly (for the fewer synchronizations, atomic operations, and other extra instructions).

Which transformation best suits a given GPU kernel depends on the properties of the kernel. We design a runtime version selector, which selects the appropriate version on the fly (without the need for offline profiling). Figure 7 outlines our selection algorithm. The algorithm uses the host side code to select either T3 or T4. It leaves the selection between T1 and T2 to the runtime execution at the device side (Figure 7(b)). The device side is also used when the host can tell the numbers of subkernels and their sizes.

The host side algorithm computes the maximum number of parent thread blocks that can be concurrently supported by the GPU and sets that number as the number of the persistent parent thread blocks. If that number is larger than the number of parent thread blocks set in the original kernel, using T3 or T4 would leave some persistent blocks idle. Hence, either T1 or T2 is suitable (the T1_T2_megakernel is a code with both T1 and T2 put together for runtime selection on the device; exemplified in Figure 10 (e)). Otherwise, the algorithm checks whether the sizes of the subkernels, in terms of the number of children thread blocks, are highly imbalanced: The standard deviation of the sizes is greater than the mean size or not (line 5 in Figure 7(a)). If so, global balancing is important and hence the T1_T2_megakernel is chosen. Otherwise, it checks whether more than half of threads launch subkernels. If so, the simplest thread-level subtask assignment (T4) is a good choice as most threads will be busy and the runtime control overhead is minimized (line 7 in Figure 7(a)). Otherwise, the better load balance offered by T3 gives the advantage and it is chosen (line 9 in Figure 7(a)).

The T1_T2_megakernel is chosen for imbalanced child kernel launches and the cases when the host does not have enough info on the size or number of calls of the subkernel. The algorithm in Figure 7(b) selects between T1 and T2 at the runtime of the GPU kernel of interest. Recall that the differences between them is that although both support even assignments to all parent thread blocks, T1 treats each subtask as the assignment unit, while T2 treats the whole set of subtasks of a subkernel as the assignment unit. The algorithm in Figure 7(b) uses the runtime revealed subkernel numbers and sizes to compute the number of extra subtasks the most loaded persistent thread block is subject if T2 is applied, compared to the load each persistent thread block has in T1 (line 3 in Figure 7(b)). It estimates the overhead of the extra control instructions in T1 (stored in "extraOverhead_T1"). It then compares the overhead of the extra subtasks in T2 and the extra control overhead in T1, and selects the more efficient one (lines 4 to 7 in Figure 7(b)).



```
1   persistTBs = maxActiveTBsAllowed();
2   if (persistTBs > orgParTB)
3       choose T1_T2_megaKernel;
4   else
5       if (CTB_std < CTB_mean)
6          if (over half threads launch subkernels)
7             choose T4;
8          else
9             choose T3;
10      else //CTB_std ≥ CTB_mean or uncertain
11         choose T1_T2_megaKernel;
```

(a) Host side

```
1   subtasksPerPTB_T1 = ttlSubtasks/PersistTBs;
2   maxSubtasks_T2 = findMaxSubkernelSize();
3   extraSubtasks = maxSubtasks_T2 - subtasksPerPTB_T1;
4   if (extraSubtasks * subtaskLength > extraOverhead_T1)
5     choose T1;
6   else
7     choose T2;
```

(b) GPU side

Figure 7: Algorithm for transformation selection.

Figure 8 offers an example. The bars in broken lines show the number of subtasks each persistent thread block (PTB) gets in T1. The black bars show the numbers of assigned subtasks that the algorithm estimates for each PTB in T2. The third PTB is the mostly loaded, having (88-47=43) extra subtasks. The algorithm estimates the cost of them by multiplying 43 with the estimated length of a subtask (in number of instructions). If it is larger than the estimated extra control overhead of T1, T1 is used; otherwise, T2 is used.

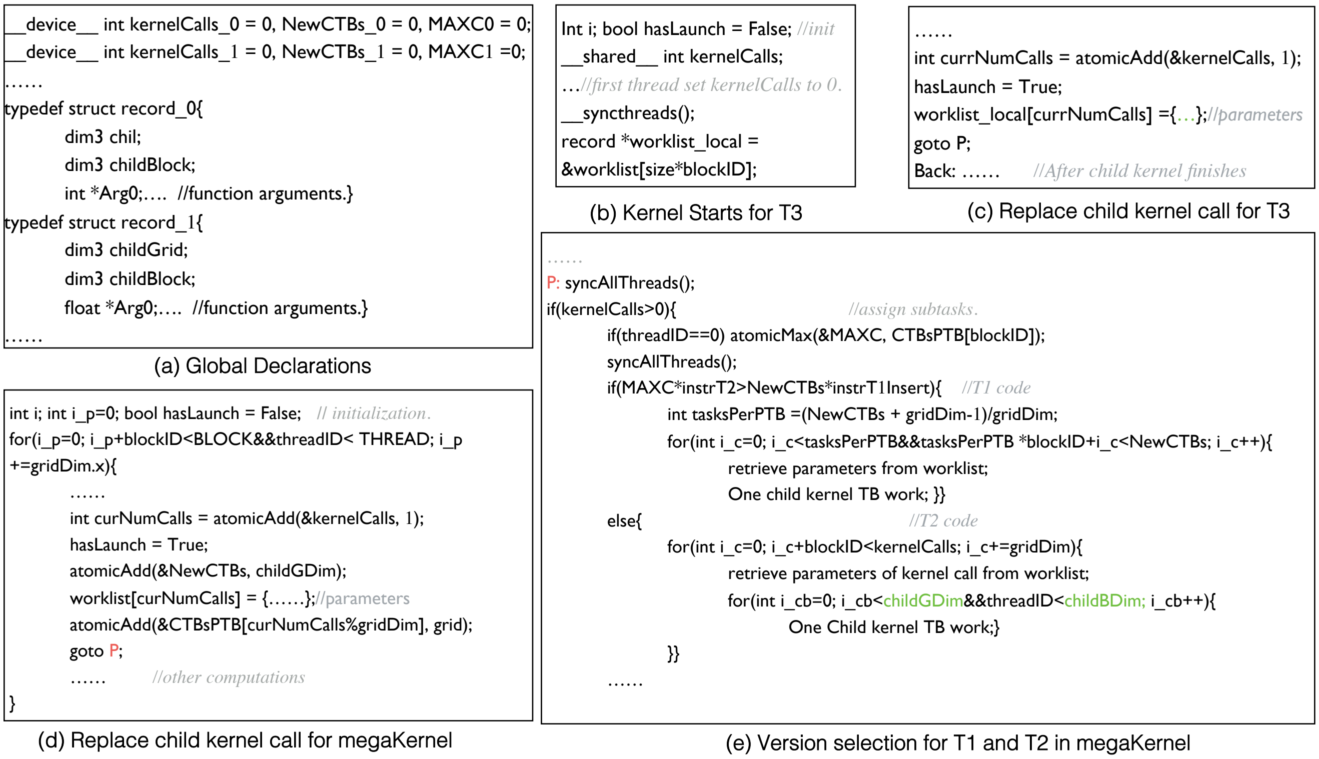
Device side selection has more accurate information on the subtasks. In principle, one could put all four versions into a megakernel for selection at the device side. The disadvantages are the increased code size, and the register and shared memory pressure. Our design favors T1 and T2 slightly for their advantages in load balancing.

4. IMPLEMENTATION

Figure 9 shows the high level structure of *free launch*, which contains two components: a source-to-source compiler for revealing the launching patterns of the program and staging the code for runtime adaptation, and an online selector which takes the launching patterns and launching profiles as inputs to find out the best version at runtime.

We implement the source-to-source compiler based on Cetus [9]. The compiler takes the following steps to generate the four versions of code for four different transformations, along with the version selection code.

- Find out how many child kernels will be launched by one thread at most and add codelet shown in Figure 10(a) to the global declarations for each. For each child kernel call, find out how many parameters need to be stored and create the needed data structure.



```
__device__ int kernelCalls_0 = 0, NewCTBs_0 = 0, MAXC0 = 0;
__device__ int kernelCalls_1 = 0, NewCTBs_1 = 0, MAXC1 =0;
……
typedef struct record_0{
        dim3 chil;
        dim3 childBlock;
        int *Arg0;….  //function arguments.}
typedef struct record_1{
        dim3 childGrid;
        dim3 childBlock;
        float *Arg0;….  //function arguments.}
……
```

(a) Global Declarations

```
Int i; bool hasLaunch = False; //init
__shared__ int kernelCalls;
…//first thread set kernelCalls to 0.
__syncthreads();
record *worklist_local =
&worklist[size*blockID];
```

(b) Kernel Starts for T3

```
……
int currNumCalls = atomicAdd(&kernelCalls, 1);
hasLaunch = True;
worklist_local[currNumCalls] ={…};//parameters
goto P;
Back: ……      //After child kernel finishes
```

(c) Replace child kernel call for T3

```
int i; int i_p=0; bool hasLaunch = False;   // initialization.
for(i_p=0; i_p+blockID<BLOCK&&threadID< THREAD; i_p
+=gridDim.x){
        ……
        int curNumCalls = atomicAdd(&kernelCalls, 1);
        hasLaunch = True;
        atomicAdd(&NewCTBs, childGDim);
        worklist[curNumCalls] = {……};//parameters
        atomicAdd(&CTBsPTB[curNumCalls%gridDim], grid);
        goto P;
        ……        //other computations
}
```

(d) Replace child kernel call for megaKernel

```
……
P: syncAllThreads();
if(kernelCalls>0){                          //assign subtasks.
        if(threadID==0) atomicMax(&MAXC, CTBsPTB[blockID]);
        syncAllThreads();
        if(MAXC*instrT2>NewCTBs*instrT1Insert){    //T1 code
                int tasksPerPTB =(NewCTBs + gridDim-1)/gridDim;
                for(int i_c=0; i_c<tasksPerPTB&&tasksPerPTB *blockID+i_c<NewCTBs; i_c++){
                        retrieve parameters from worklist;
                        One child kernel TB work; }}
        else{                                        //T2 code
                for(int i_c=0; i_c+blockID<kernelCalls; i_c+=gridDim){
                        retrieve parameters of kernel call from worklist;
                        for(int i_cb=0; i_cb<childGDim&&threadID<childBDim; i_cb++){
                                One Child kernel TB work;}
                }}
        ……
```

(e) Version selection for T1 and T2 in megaKernel

Figure 10: A codelet for code generations. threadID, blockID and gridDim are changed to one dimension value.

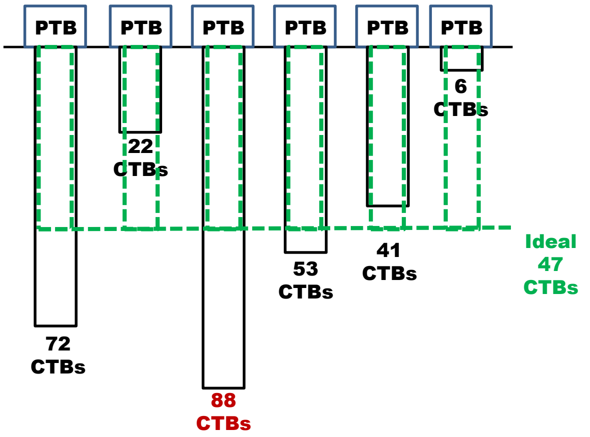


Figure 8: An example of subtasks assignments by T1 (in broken lined bars) and T2 (in solid lined bars).

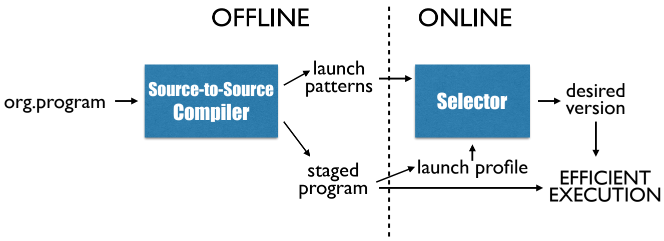
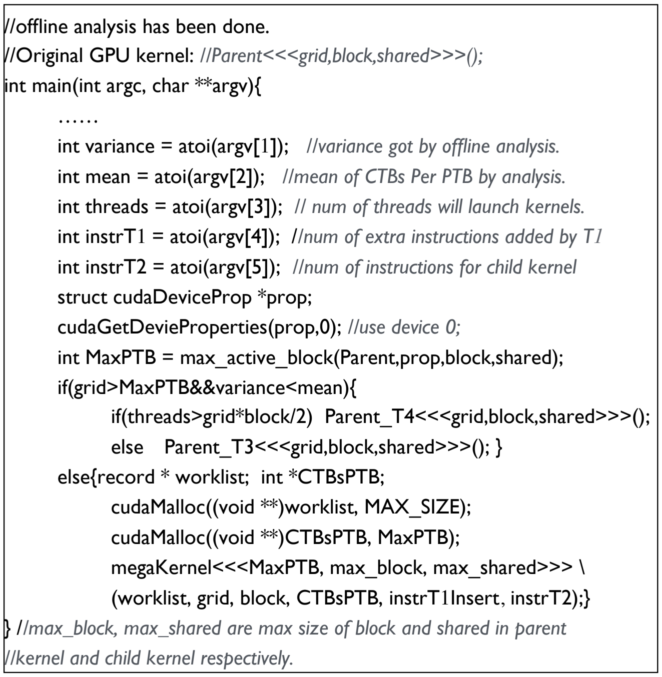


Figure 9: High level structure of *free launch*.

- Statically analyze the launching patterns, based on section 3 to determine whether T3 or T4 code needs to be generated (e.g., if the program is irregular, T3 and T4 need not to be generated). To generate different versions of transformations, we have different strategies. For T4 code to be generated, We use the same technique as MCUDA [10] does to generate a sequential code for a parent thread to run to do the subtasks. For T3 code to be generated, in the parent kernel, the codelet in Figure 10 (b) will be added at the beginning, and the codelet in Figure 10(c) will be added at the position where the child kernel is launched. For T1 and T2 code to be generated, we create a megakernel which will contain the code of both T1 and T2. The preparation for recording parameters and online profiles will be done in code shown in Figure 10(d). To choose T1 code or T2 code in the megakernel, we add code in Figure 10(e) to calculate T1 and T2 performance based on the profiling data we get in Figure 10 (d).

- Transform the host code to enable the selection between T3 code and T4 code if necessary, as illustrated in Figure 11.



```
//offline analysis has been done.
//Original GPU kernel: //Parent<<<grid,block,shared>>>();
int main(int argc, char **argv){
        ……
        int variance = atoi(argv[1]);  //variance got by offline analysis.
        int mean = atoi(argv[2]);  //mean of CTBs Per PTB by analysis.
        int threads = atoi(argv[3]);  // num of threads will launch kernels.
        int instrT1 = atoi(argv[4]);  //num of extra instructions added by T1
        int instrT2 = atoi(argv[5]);  //num of instructions for child kernel
        struct cudaDeviceProp *prop;
        cudaGetDevieProperties(prop,0); //use device 0;
        int MaxPTB = max_active_block(Parent,prop,block,shared);
        if(grid>MaxPTB&&variance<mean){
                if(threads>grid*block/2)  Parent_T4<<<grid,block,shared>>>();
                else    Parent_T3<<<grid,block,shared>>>(); }
        else{record * worklist;  int *CTBsPTB;
                cudaMalloc((void **)worklist, MAX_SIZE);
                cudaMalloc((void **)CTBsPTB, MaxPTB);
                megaKernel<<<MaxPTB, max_block, max_shared>>> \
                (worklist, grid, block, CTBsPTB, instrT1Insert, instrT2);}
} //max_block, max_shared are max size of block and shared in parent
//kernel and child kernel respectively.
```

Figure 11: A sample of runtime version selection put in the host-side code before a kernel call.

5. EVALUATION

This section evaluates the proposed launch removal technique. It gives performance comparisons on several different versions of a set of programs: the default version from benchmark suites that use no dynamic parallelism, the version that uses dynamic parallelism, and the versions produced by the various transformations of our kernel removal. These versions differ just at how the dynamic parallelism inside a GPU kernel is materialized; the top-level kernel and task management (in the host code) is the same in these versions.

5.1 Methodology

Because of the large overhead of dynamic subkernel launch—the exact problem this work tries to address, we did not find many existing benchmarks that use dynamic parallelism. We looked through several public CUDA benchmark suites and identified eight kernels of seven programs that either have already used dynamic parallelism, or have a GPU kernel which contains some loops that could be conveniently expressed in subkernel calls. For the latter ones, we then manually made code changes to replace the loops with child kernel calls.

Figure 2 lists those programs, with a brief description of the programs and the functionality of the dynamic kernel launches. The citations in the first column indicate the source of the original version of the benchmark (*SSSP* was implemented in this work, which follows the classic shortest path algorithm, and manages the parallelism in a way similar to the *BFS* in Rodinia). The rightmost column indicates whether the subkernel is regular or irregular (detailed in Section 5.2).

Table 1: Machine Description.

Name	GPU card	Processor	CUDA
K20c	NVIDIA K20c	Intel Xeon E5-1607v2	6.5
K40m	NVIDIA K40m	Intel Xeon E5-2697	7.0

Among the programs, *CCL* and *BT* already use dynamic parallelism in the original code. *SP* and *MST* come from the LoneStarGPU [1] benchmark suite (v2.0). They use double-buffered worklists for storing frontier nodes for computing efficiency. The subkernel launch (SKL) version of *SP* uses dynamic kernel calls to replace the "for(int edndx = 0; edndx < cllen; edndx++)" loop inside the kernel "decimate_2". The usage of double-buffered worklists is not affected, still used in the SKL version and all the free launch versions. Two kernels of *MST* are used in the study, one to find the minimum spanning tree, the other to verify the minimum spanning tree. Dynamic subkernel calls are used to replace the loops for processing all neighbors of a given node in the two kernels. For the graph coloring program *GC*, the dynamic parallelism is used to replace the loop for processing the adjacent nodes of a graph node in the kernel for detecting the numbers of conflicts for current colored graph. Programs *BFS* and *SSSP* both come from the Rodinia benchmark suite [11]. The original code of *BFS* uses one warp to process 32 nodes of a graph, and for each node, it uses that warp to process its neighbors in parallel. In the SKL version, each parent thread visits one node, and a subkernel call is used for processing all the neighbors of that node. The parallelism in *SSSP* is similar to *BFS*.

All subkernel calls, except those in CCL, are asynchronous. When measuring the performance of kernels that contain subkernel calls, we use the CUDA function "cudaDeviceSetLimit" to ensure that the pending launch limit "cudaLimitDevRuntimePendingLaunchCount" of GPU subkernel launches is not exceeded by the actual subkernel launches. It helps reserve enough resource on the device for subkernel launches. Without it, the slowdown brought by subkernel launches is several times larger than using it.

We run the experiments on both K20c and K40m GPU as listed in Table 1. The results on them are similar. We include the overall performance results on both of them in Figure 12. To enhance the legibility of *CCL* and *BT* results, we list their numbers in Table 3. Our detailed result discussions, without notice, concentrate on the results collected on K20c.

5.2 Overall Performance

The transformed code run correctly on all of the programs, regardless of their code complexities (atomic operations in MST_dfind2 and MST_verify, synchronization in CCL, function calls in CCL and BT kernels, etc.). Although in principle it is possible for free launch to support recursive subkernel launches (with a limit on recursion levels), none of the kernels have recursive launches. We leave the support to future work.

Table 2: Benchmarks

CCL and BT already use dynamic subkernel launches in their original version.

Benchmark	Description	Purpose of the dynamic kernel call	Regular
GC [12]	Graph Coloring	color a node's all adjacent node	Y
SP [1]	Survey Propagation	propagate to a variable's neighbours	N
BFS [11]	Breadth-first Search in a graph	determine the levels of all neighbour nodes of a given node	N
SSSP [13]	Single-Source Shortest Paths in a graph	get the minimal distance from a node to its neighbours	N
CCL [14]	Connected Component Labelling	Merge different spans	Y
BT [15]	Bezier Line Tessellation	all vertices along each bezzier line	N
MST_dfind2 [1]	find Minimum Spanning Tree	process all neighbours of a given node	N
MST_verfiy [1]	verify Minimum Spanning Tree	process all neighbours of a given node	N

Table 3: Speedups of CCL and BT over the original SKL version.

Kernels	K20c						K40m					
	T1	T2	T3	T4	auto-selection	original	T1	T2	T3	T4	auto-selection	original
CCL	1.17	0.31	0.43	0.0041	1.17	1	1.08	0.32	0.31	0.0046	1.08	1
BT	0.81	0.97	1.94	2.05	1.91	1	0.79	0.94	1.89	2	1.84	1

Overhead of Subkernel Launches.

The "original" bars in Figure 12 correspond to the original benchmarks, which, except for *CCL* and *BT*, use no subkernel calls. The up to 21X speedups shown by the "original" bars indicate the large overhead of subkernel launches. The result echos some numbers mentioned in a previous paper [6]. It is however much larger than what another recent paper reports [4]. In that work, the authors included only the cases where the amount of work for a child kernel to do is larger than some thresholds. The threshold values were manually determined for each benchmark. The overhead of subkernel launches is more prominent when the amount of work for a child kernel to do is small. Our objective is to automatically address the overhead in all cases and hence uses no such manually customized thresholds.

The "auto-selection" bars in Figure 12 correspond to the results of our method equipped with the automatic version selection (all runtime overhead is counted). By removing the subkernel launching overhead, the transformed results outperform the version with subkernel calls by 36X on average. The rightmost column of Table 4 shows the selected versions. Among them, T2 is the most popular choice, T3 is never selected. Figure 12 shows that the auto-selector selects the best version for all the benchmarks; the selection overhead throttles the speedups by about 10% on average.

Speedups over the Original.

From Figure 12, we can see that the results from "auto-selection" also outperform the "original" substantially: 3.2X on K20c and 3.4X on K40m on average. When excluding the two programs (CCL and BT) that use subkernel calls in their original versions, the average speedups over the "original" are still about 3X. The main reason is the much better load balance brought by the free launch version. It is shown by the right columns in Table 4 under "Max subtasks by a PTB". The columns report the maximum number of subtasks assigned to a parent thread block in the four free launch versions and in the original version. For a given kernel, the total number of subtasks is the same for all the versions. So, the lower the number in a column is, the more balanced the workload is in that version. The maximum numbers of subtasks in the "original" version are orders of magnitude larger than the free launch versions (especially T1 and T2).

We take SP as an example for a more detailed discussion. At the core of SP, it tries to solve a SAT problem. Recall that the SKL version of SP uses dynamic kernel calls to replace the "for" loop inside the kernel "decimate_2", which propagates a boolean value to a variable's neighbours (i.e., clauses containing that variable). The generated T2 version shows 2.2X speedup compared to the original version (the default input "random-16800-4000-3.cnf" is used). In the execution of the original version, the kernel has 65 thread blocks and each has 384 threads, but only 40 threads in the first thread block actually work on the "for" loop and other threads are idle. The numbers of iterations of the loop executed by the threads frequently differ. In T2, the work in the 40 loops is handled by the 65 parent thread blocks with a much improved load balance. As Table 4 shows, the maximum numbers of subtasks by a PTB are 240 for the "original" and 4 for T2.

The "R" fields in three columns in Table 4 report the number of registers used in the GPU kernels. On several benchmarks due to the use of megakernel and inserted instructions, the free launch versions use slightly more registers than the parent kernels do in the version that uses subkernel calls. There is some slight decrease in register usage on *CCL* and *BT* due to different register reuses. The other columns of Table 4 show more detailed information on the kernels, including the grid and thread block dimensions, and the numbers of child kernel launches.

We next discuss the benchmarks further by grouping them into two categories.

Regular Cases.

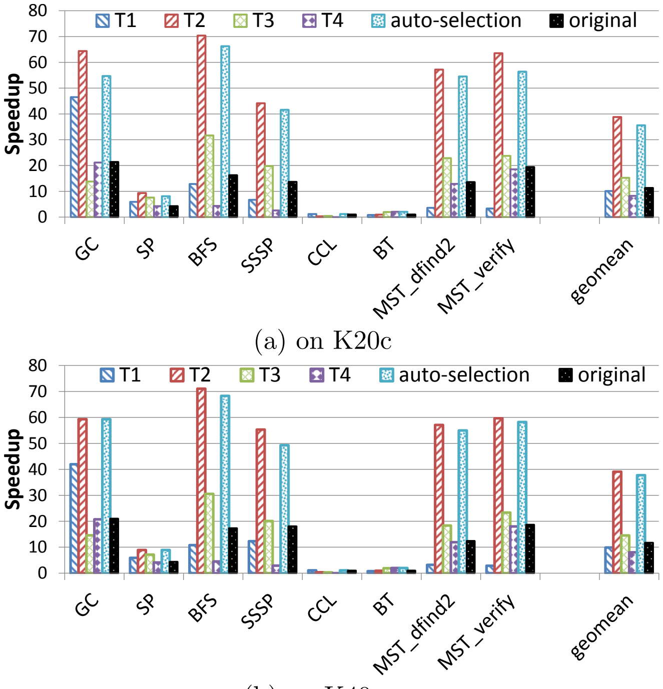
Benchmarks GC and CCL are regular cases whose child kernel launches' patterns can be analyzed statically during compile time.

For GC, the SKL version launches a kernel with 4 thread blocks and each contains 256 threads. Each thread launches a child kernel. As a result, there are

Table 4: Details of each kernel.

"Concurrency": the number of subtasks a parent thread block does concurrently in the free launch versions;
[...]: parameters on the grid and block dimensions of a kernel;"R": the number of registers in a kernel

Kernels	Parent Kernel	Child Kernel	Child kernel launches	Concurrency	Max subtasks by a PTB					Selected version
					T1	T2	T3	T4	org	
GC	[4,256],R:18	[2,32],R:8	1024	8	20	20	64	512	376	T2,R:20
SP	[65,384],R:27	[unknown,32],R:8	40	12	4	4	40	320	240	T2,R:27
BFS	[10,1024],R:18	[unknown,32],R:8	1000	32	99	139	1279	32768	32032	T2,R:26
SSSP	[10,1024],R:20	[unknown,32],R:12	1001	32	101	139	1311	32768	32032	T2,R:32
CCL	[1,2],R:45	[4,256],R:6	2	1	1	4	8	1024	NA	T1,R:37
BT	[400,64],R:26	[unknown,32],R:18	25600	2	64	64	64	64	NA	T4,R:23
MST_dfind2	[4,1024],R:40	[unknown,32],R:40	3947	32	345	358	1143	7168	6432	T2,R:40
MST_verfiy	[4,1024],R:40	[unknown,32],R:40	3947	32	345	357	1143	7168	6432	T2,R:40



(a) on K20c

(b) on K40m

Figure 12: Speedups over the version that uses subkernel launches. The "original" bars show the performance of the original version of the code without using subkernel launches (*CCL* and *BT* already uses subkernel launches in their original version, hence excluded from that category). Table 3 shows the detailed results of *CCL* and *BT*.

1024 child kernel launches. The selected transformed version launches the maximum number of thread blocks, which is 8*13 instead of 4, for the parent GPU kernel. Since the configuration of each child kernel launch can be known during compile time, the transformed code doesn't need inter-block synchronization to record each configuration. The assignments in T1 and T2 are actually the same—that is, each parent block handles at most 20 subtasks as Table 4 shows, which helps T2 achieve a 64X speedup. However, for T1, it has extra overhead in getting the configuration of each subtask; hence a lower speedup (47X). For T3, the maximum number of subtasks handled by one parent block is 64. For T4, one thread would process the entire work of child kernel, which consists of at most 47 children threads' work. For T3 and T4, the occupancy is low. The selected version is T2, which provides a 55X speedup including the analysis overhead.

For CCL, in the original code execution, the parent kernel has only 1 thread block with 2 threads, and each thread launches a child kernel with 4 thread blocks (each contains 256 threads). The T1 transformation creates eight persistent thread blocks, and each parent thread block processes one subtask. The overhead is trivial here since there are only 8 subtasks to assign. T1 brings the best performance with a 1.17x speedup. The other transformations result in poor load balance on it. T1 is chosen.

Irregular Cases.

In benchmarks BT, SP, BFS, SSSP, MST_dfind2 and MST_verify, the static code analysis cannot determine which parent threads make subkernel calls and how many subtasks a subkernel contains.

BT is an opposite example compared with GC. The parent kernel in the SKL version has occupied all resources on all GPU streaming multiprocessors. So all created children thread blocks would compete for resources with parent thread blocks; launching children kernels does not help improve the occupancy much but increases a lot of launching overhead and scheduling overhead. For the same reason, T1 and T2 do not help; their overhead causes some slowdown. T4 works the best by letting each thread do all tasks it creates with no synchronization overhead, yielding a 2.05X speedup. T4 is chosen.

For SP, the SKL version launches a kernel with the maximum number of parent thread blocks. Inside the kernel, 40 child kernels get launched; which thread launches a kernel depends on some runtime conditions. T1 and T2 have 65 persistent thread blocks. Their subtask assignments ensure that one persistent block processes at most one subtask, giving a superior load balance. T1 gives a 5.89X speedup, while T2 gives a 9.31X speedup for its lower runtime overhead. The local assignments by T3 and T4 result in some imbalance with the maximum number of subtasks of one parent block reaching 40 and 320 respectively.

For other benchmarks, in the same vein, the four transformations give different speedups. The auto se-

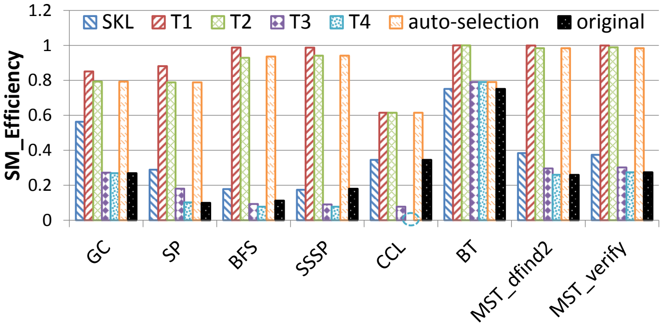


Figure 13: SM efficiency.

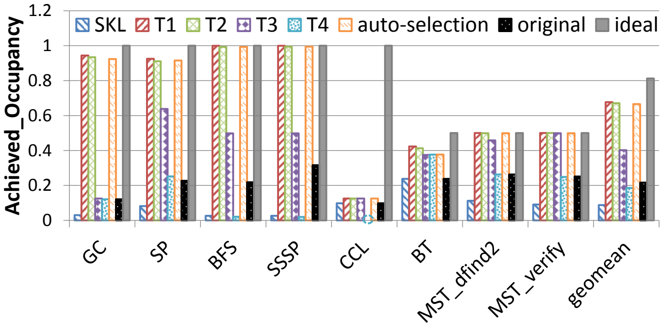


Figure 14: Achieved SM occupancy.

lector selects version T2 for those programs because the workload balance is good without extra overhead for retrieving parameters for each kernel launch.

On GC, SP, MST_dfind2, and MST_verify, the original versions are similar to the T4 version, and hence give performance similar to T4. On BFS and SSSP, the original versions differ from T4 in that they use a warp to process the neighbors of a node in parallel, while T4 processes the neighbors of a node through only the parent thread that handles that node. As a result, on these two benchmarks, T4 runs slower than the original.

The impact of the transformation to the whole application depends on how much the kernel weighs in the whole application, which varies with program inputs for many of them. For the set of experiments we conducted, the speedups of the whole applications range from 1.05X to 1.3X with an average of 1.18X.

5.3 Analysis through Hardware Counters

We conduct some deeper analysis of the results through nvprof [15], the profiling tool provided by NIVIDIA. Figure 13 shows the SM efficiency, which is the percentage of time at least one warp is active on an SM, averaged over all SMs on the GPU. Figure 14 shows the achieved occupancy of an SM, which is ratio of the average active warps per active cycle to the maximum number of warps supported on an SM. The ideal-max means the theoretical occupancy calculated by the usage of resources (registers, shared memory, etc.). The tool fails in getting the *SM efficiency* and *achieved occupancy* of CCL.

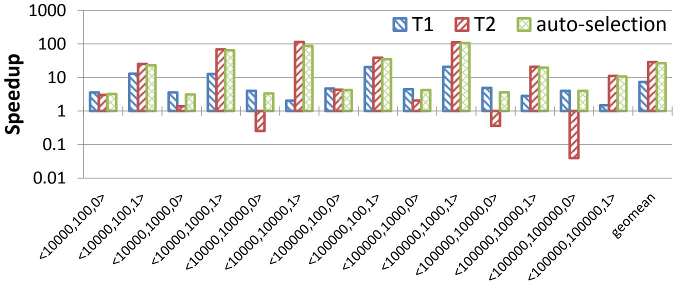


Figure 15: Speedup of BFS across different inputs

In Figure 13, the kernels with dynamic parallelism have an average 0.3 SM efficiency due to the expensive launching overhead and block scheduling overhead, which means most of warps are not active in an active cycle. It helps explain the low *achieved occupancy* (average 0.067) in Figure 14.

For T1, it always has the highest *SM efficiency* (average 0.915) and *achieved occupancy* (average 0.68) compared to the other five versions. There are two reasons: the first is that there is no subkernel launching overhead and the kernel has the maximum number of active thread blocks; the second is that the extra code added for T1 to retrieve the configuration of subtasks keeps the SM more likely to be busy. However, T1 doesn't always provide the best performance due to the overhead of synchronizations and parameter retrieval for each child kernel launch.

T2 achieves a similar *SM efficiency* (average 0.88) and *Achieved occupancy*(average 0.67) as T1 does. Thanks to its low overhead, it outperforms T1. T1 and T2 have near 100% *SM efficiency* in Figure 13 and the ideal-max occupancy(0.81) for SM *Achieved occupancy* in Figure 14. The only exception is CCL; its low *achieved occupancy* (0.125) is because the number of subkernels launches by parent kernel is small and there is no need to create the maximum number of persistent threads.

For T3 and T4, they work well when SM *achieved occupancy* of original parent kernel is very high and the amount of subtasks created by each parent thread block is similar. BT is such an example, as we explained in section 5.2.

5.4 Sensitivity to Different Inputs

The benefits of free launch may vary across different program inputs. We choose one kernel, BFS, to study the variations across different inputs. We consider only two transformation versions (T1,T2) as the BFS kernel is irregular.

Figure 15 shows the speedup of BFS on different inputs and iterations over the SKL version. The axis shows the different inputs, with each characterized in a tuple, showing the number of nodes, average degree of a node, and the level that the kernel is looking for. For example, <10000,100,0> means the kernel is searching for nodes in level 0 and then assigns 1 to the levels of their children nodes, the number of graph nodes is 10000, and the average degree of each node is 100. The speedups by the auto-selected free launch transformation show vary between 3X and 87X. For all graphs, at level 0, T1 always achieves the best performance. The reason is that only one child kernel is launched and the superior load balance of T1 over T2 plays an important role. At level 1 for all graphs, T2 achieves the best performance because at this level, many parent threads launch subkernels (over 1000 threads for <10000,1000,1>). T2 can also get a good load balance. For T1, it can also achieve the best workloads balance. However, because the overhead of retrieving parameters of child kernel for each subtask is high, its performance is not as good as T2.

6. RELATED WORK

There have been many studies published in optimizing GPU program performance. In this section, we focus on the work closely related with dynamic parallelism in GPU.

To help expose enough parallelism to efficiently use GPU, NVIDIA introduced dynamic parallelism [2], where threads can launch subkernels dynamiclly. Using the feature, CUDA SDK [15] achieves a better speed on QuickSort. DiMarco and others show the performance impact of the feature on clustering algorithms [16].

A drawback of subkernel launch is the high launching overhead. In a recent work, Wang and Yalamanchili have characterized and analyzed subkernel launches in unstructured GPU applications [4] and have proposed a new method called "Dynamic Thread Block Launch" [5], which is a lightweight execution mechanism to reduce the high launching overhead. The work shows good potential in improving the efficiency, but requires extra hardware extensions. Yang and others develop a compiler called "CUDA-NP" [6], which tries to exploit nested parallelism by creating a large number of GPU threads initially and use control flows to activate different numbers of threads for different code sections. The parallelism is limited to the number of threads in a thread block and has shown to work on nested loops only. The *SM-centric transformation* recently proposed by Wu and others [17] offers a simple approach to controlling the placement of GPU tasks on GPU streaming multiprocessors, and shows significant benefits for enhancing data locality across GPU threads and the co-running performance of multiple kernels.

On optimizing irregular computations on GPU, Zhang and others have developed G-Streamline, which removes non-coalesced memory references and thread divergences through a runtime thread-to-data remapping [18, 19]. Wu and others extend the work by proving the NP-completeness of the remapping problem and proposing two new optimizations to the irregular computations [20]. Other related studies concentrate on the memory performance of irregular computations on GPU [21, 22, 23, 24]. A software framework named PORPLE has been recently proposed to dynamically finds and realizes the best placements of data on the various types of memory on GPU [25]. Some other work [26, 27] contributes benchmarks or quantitative studies for characterizing and analyzing irregular applications.

Manson and others have identified some possible correctness concerns of thread inlining in a study on Java memory models [28]. GPU memory models are currently not strictly defined. We did not find issues with the out-of-thin-air values in the proposed transformations.

7. CONCLUSION

In this paper, we have presented *free launch*, the first pure software solution that removes the large overhead of GPU dynamic subkernel launches from a program through four kinds of automatic *launch removal* and an *adaptive version selection* technique. The technique demonstrates the promise for overcoming shortcomings of prior solutions for efficiently supporting dynamic parallelism on GPU. With it, programmers can enjoy the convenience and expressiveness of subkernel launches for GPU programming, while avoiding the associated large cost and achieving a superior load balance. The technique even outperforms manual worklist-based support of dynamic parallelism by 3X in terms of average speedups. By turning subkernel launch into a programming feature independent of hardware support, *free launch* makes subkernel launch even possible to use beneficially on GPUs that do not offer hardware support of subkernel launches.

Acknowledgement

We thank the Micro'15 reviewers for their helpful suggestions. This material is based upon work supported by the DOE Early Career Award, the National Science Foundation (NSF) (under Grants No. 1455404, 1525609, and Career Award), IBM CAS Fellowship, and Google Faculty Award. The experiments benefit from the GPU devices donated by NVIDIA. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DOE, NSF, IBM, Google, or NVIDIA.

References

[1] M. Kulkarni, M. Burtscher, K. Pingali, and C. Cascaval, "Lonestar: A suite of parallel irregular programs," in *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software*, 2009.

[2] S. Jones, "Introduction to dynamic parallelism," in *Nvidia GPU Technology Conference*, (San Jose, CA), May 2012.

[3] "OpenCL." `http://www.khronos.org/opencl/`.

[4] J. Wang and S. Yalamanchili, "Characterization and analysis of dynamic parallelism in unstructured gpu applications," in *2014 IEEE International Symposium on Workload Characterization*, October 2014.

[5] J. Wang, N. Rubin, A. Sidelnik, and S. Yalamanchili, "Dynamic thread block launch: A lightweight execution mechanism to support irregular applications on gpus," in *Proceed-*

ing of the 42nd Annual International Symposium on Computer Architecuture (ISCA-42), June 2015.

[6] Y. Yang and H. Zhou, "Cuda-np: Realizing nested thread-level parallelism in gpgpu applications," *SIGPLAN Not.*, vol. 49, pp. 93–106, Feb. 2014.

[7] J. Kim and C. Battern, "Accelerating irregular algorithms on gpgpus using fine-grain hardware worklists," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014.

[8] 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.

[9] S. Lee, T. Johnson, and R. Eigenmann, "Cetus - an extensible compiler infrastructure for source-to-source transformation," in *In Proceedings of the 16th Annual Workshop on Languages and Compilers for Parallel Computing (LCPC)*, pp. 539–553, 2003.

[10] J. Stratton, S. Stone, and W. Hwu, "MCUDA: An Efficient Implementation of CUDA Kernels for Multi-Core CPUs," in *Languages and Compilers for Parallel Computing: 21th International Workshop (LCPC)*, 2008.

[11] 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 *GPGPU*, 2010.

[12] 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, (New York, NY, USA), pp. 297–298, ACM, 2011.

[13] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms.* MIT Press and McGraw - Hill, 2002.

[14] 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, (New York, NY, USA), pp. 253–264, ACM, 2015.

[15] "NVIDIA CUDA." `http://www.nvidia.com/cuda`.

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

[17] B. Wu, G. Chen, D. Li, X. Shen, and J. Vetter, "Enabling and exploiting flexible task assignment on gpu through sm-centric program transformations," in *Proceedings of the ACM International Conference on Supercomputing*, 2015.

[18] E. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. Shen, "On-the-fly elimination of dynamic irregularities for gpu computing," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

[19] E. Z. Zhang, Y. Jiang, Z. Guo, and X. Shen, "Streamlining gpu applications on the fly," in *Proceedings of the ACM International Conference on Supercomputing (ICS)*, pp. 115–125, 2010.

[20] B. Wu, Z. Zhao, E. Z. Zhang, Y. Jiang, and X. Shen, "Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on gpu," in *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2013.

[21] P. Carribault, A. Cohen, and W. Jalby, "Deep jam: conversion of coarse-grain parallelism to instruction-level and vector parallelism for irregular applications," in *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, (St. Louis, MO), 2005.

[22] M. Kulkarni, M. Burtscher, R. Inkulu, K. Pingali, and C. Casçaval, "How much parallelism is there in irregular applications?," in *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, (New York, NY, USA), pp. 3–14, ACM, 2009.

[23] S. Kim, H. Han, and K. Choe, "Region-based parallelization of irregular reductions onÂăexplicitly managed memory hierarchies," *Journal of Supercomputing*, 2009.

[24] M. Burtscher, M. Kulkarni, D. Prountzos, and K. Pingali, "On the scalability of an automatically parallelized irregular application," in *Languages and Compilers for Parallel Computing: 21th International Workshop, LCPC 2008, Edmonton, Canada, July 31 - August 2, 2008, Revised Selected Papers*, (Berlin, Heidelberg), pp. 109–123, Springer-Verlag, 2008.

[25] G. Chen, B. Wu, D. Li, and X. Shen, "Porple: An extensible optimizer for portable data placement on gpu," in *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture*, 2014.

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

[27] M. Burtscher, R. Nasre, and K. Pingali, "A quantitative study of irregular programs on gpus," in *IISWC*, 2012.

[28] J. Manson, W. Pugh, and S. Adve, "The java memory model," in *Proceedings of the ACM SIGPLAN Conference on Principles of Programming Languages (POPL)*, 2005.