Taming Irregular Applications via Advanced Dynamic Parallelism on GPUs

Jing Zhang
Dept. of Comp. Sci.
Virginia Tech
zjing14@vt.edu

Ashwin M. Aji
AMD Research
Advanced Micro Devices, Inc.
aaji@amd.com

Michael L. Chu
AMD Research
Advanced Micro Devices, Inc.
mike.chu@amd.com

Hao Wang
Dept. of Comp. Sci.
Virginia Tech
hwang121@vt.edu

Wu-chun Feng
Dept. of Comp. Sci.
Virginia Tech
wfeng@vt.edu

ABSTRACT

On recent GPU architectures, dynamic parallelism, which enables the launching of kernels from the GPU without CPU involvement, provides a way to improve the performance of irregular applications by generating child kernels dynamically to reduce workload imbalance and improve GPU utilization. However, in practice, dynamic parallelism does not improve performance due to high kernel launch overhead and low child kernel occupancy. Consequently, most existing studies focus on mitigating the kernel launch overhead. As the kernel launch overhead has decreased due to algorithmic redesigns and hardware architectural innovations, the organization of subtasks to child kernels becomes a new performance bottleneck.

We present an in-depth characterization of existing software approaches for dynamic parallelism optimizations on the latest GPUs. We observe that current approaches of subtask aggregation, which use the "one-size-fits-all" method by treating all subtasks equally, can under-utilize resources and degrade overall performance, as different subtasks require various configurations for optimal performance. To address this problem, we leverage statistical and machine-learning techniques and propose a performance modeling and task scheduling tool that can (1) analyze the performance characteristics of subtasks to identify the critical performance factors, (2) predict the performance of new subtasks, and (3) generate the optimal aggregation strategy for new subtasks. Experimental results show that our approach with the optimal subtask aggregation strategy can achieve up to a 1.8-fold speedup over the existing task aggregation approach for dynamic parallelism.

CCS CONCEPTS

• **Computing methodologies → Concurrent computing methodologies**; • **Software and its engineering → Concurrent programming languages**;

KEYWORDS

Dynamic parallelism, irregular applications, performance modeling, GPU

ACM Reference Format:
Jing Zhang, Ashwin M. Aji, Michael L. Chu, Hao Wang, and Wu-chun Feng. 2018. Taming Irregular Applications via Advanced Dynamic Parallelism on GPUs. In *Proceedings of CF '18: Computing Frontiers Conference (CF '18).* ACM, Ischia, Italy, 9 pages. https://doi.org/10.1145/3203217.3203243

1 INTRODUCTION

General-purpose graphics processing units (GPGPUs) are widely used to accelerate a variety of applications in different domains. Since GPUs are ideally suited to applications with regular computations and memory access patterns, it is challenging to map irregular applications (e.g., graph algorithms, sparse linear algebra, and bioinformatics algorithms) on a GPU. Dynamic parallelism, supported by both CUDA [5] and OpenCL [2], allows a GPU kernel to directly launch other GPU kernels from the device and without involvement of the CPU. This feature can potentially improve the performance of irregular applications by reducing workload imbalance between threads, thereby improving both parallelism and memory utilization [27]. For example, during kernel execution, if some GPU threads have more work than others, new child kernels can be spawned to process these subtasks from the overloaded threads. However, the efficiency of dynamic parallelism is limited by two issues: (1) high overhead of kernel launching, especially when a large number of child kernels are needed for subtasks, and (2) low occupancy, especially when subtasks correspond to tiny kernels that under-utilize the computational resources of GPUs.

To address these two issues in dynamic parallelism, multiple approaches [11, 17, 19, 25, 26, 30] have been proposed in hardware and software. They mainly use the techniques of subtask aggregation, which consolidates small child kernels into larger kernels, hence reducing the number of kernels and increasing the GPU occupancy. However, with the kernel launch overhead continuing to decrease on the latest GPU architectures, the "one-size-fits-all" approach in these existing studies, where subtasks are aggregated into a single kernel, cannot deliver good performance because those subtasks launched by dynamic parallelism generally require *different* optimizations and configurations. As a consequence, the organization of subtasks to child kernels becomes more critical to overall performance, and an adaptive strategy of subtask aggregation that provides differentiated optimizations for subtasks with different characteristics may satisfy dynamic parallelism on the latest GPUs.

However, it is non-trivial to determine the optimal aggregation strategy for subtasks at runtime for different GPU architectures. To provide a simple system-level solution, we propose a performance modeling and task scheduling tool for subtasks in dynamic parallelism to generate the optimal aggregation strategy. Our tool collects the values of a set of GPU performance counters with sampling data and then leverages a few statistical and machine learning tools to build the performance model incrementally: (1) At the performance-analysis phase, we apply statistical analysis on GPU performance counters to identify the most influential performance factors, which can provide hints of performance-critical characteristics and performance bottleneck of subtasks. (2) At the performance-prediction phase, we build an analytical model based on the most influential performance factors and estimate the performance of new subtasks. (3) At the task-scheduling phase, our adaptive subtask aggregation strategy optimally groups the subtasks depending on the resource utilization, aggregation overhead, and kernel launch overhead, and subsequently launches the parent kernel. Compared to the "1-to-1" launching in default implementations of dynamic parallelism, where one subtask is scheduled to one child kernel and the "N-to-1" launching in previous research, where all subtasks are scheduled to an execution entity (e.g., all subtasks to a kernel or workgroup or wavefront), our "N-to-M" launching mechanism provides the most adaptability and fully utilizes GPU resources. Our paper makes the following contributions:

- An in-depth characterization of existing subtask aggregation approaches for dynamic parallelism.
- A performance model to identify the most critical performance factors and characteristics of subtasks that affect the performance and configurations of subtasks.
- The design of a subtask aggregation model based on the performance model.
- Experimental results with irregular applications and datasets on the latest GPU architectures showing that our optimal subtask aggregation strategy can achieve up to 1.8-fold speedup over the existing subtask aggregation approach.

2 BACKGROUND AND MOTIVATION

In this section, we briefly introduce current GPU architectures, GPU programming models, and dynamic parallelism in GPUs. We then introduce the statistical and machine-learning (ML) techniques for performance modeling and prediction.

2.1 GPU Architecture

Here, we take the latest AMD's GPU architecture codenamed "Vega" [6] as an example to introduce GPU architectures. Figure 1 illustrates that the AMD Vega GPU features up to 64 compute units (CUs), where each CU contains 4 SIMD units, dedicated L1 cache, and local memory. All these CUs are organized into shader engines (SEs), and share L2 cache and global memory. During the execution, threads in a kernel will be distributed across CUs, and threads on each CU will be divided into 64-thread wavefronts as basic execution units, and processed by SIMD units concurrently. If threads in a wavefront take different execution paths, it will cause control flow divergence and low SIMD utilization. If threads within a wavefront access non-consecutive memory addresses, non-coalesced memory accesses occur, resulting in memory divergence, which increases memory latency.

NVIDIA GPUs [3] have Stream Multiprocessors (SMX) grouped into Graphics Processing Clusters (GPCs). Each SMX has dedicated local memory, called shared memory, and L1 cache. L2 cache and memory controllers are shared across SMXs. Threads on an SMX will be grouped into 32-thread warps to be executed by SIMD units, just like wavefronts on AMD GPUs.

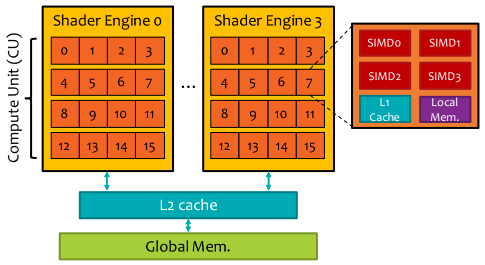


Figure 1: AMD GPU Architecture Codenamed "Vega"

2.2 GPU Programming Models

CUDA and OpenCL. CUDA [5] is a programming model developed by NVIDIA. In CUDA, a program is implemented by a set of kernels. In each kernel, threads are organized into thread blocks and distributed to SMXs. OpenCL [2] is another widely used GPU programming model, supported by AMD, NVIDIA, and Intel. Similar to CUDA, threads in a kernel are grouped into workgroups, and each workgroup will be processed on a compute unit (CU) or SMX.

ROCm. ROCm (Radeon Open Compute platform) [7] is an open-source platform created by AMD for GPU computing. ROCm supports multiple programming languages, such as HCC C++, OpenCL and HIP. The recent ROCm runtime API (version 1.6) allows the end-users to provision individual CUs for a specific kernel execution. The API enables us to explore how the pattern of resource usage and sharing affects the performance of subtasks when we vary both the number of CUs as well as the scheduling mechanism.

ATMI. The Asynchronous Task and Memory Interface (ATMI) [4] is a runtime framework and programming model for heterogeneous CPU-GPU systems. It provides a uniform API to create task graphs on both CPUs and GPUs, and enables task launching from both CPU and GPU with the same application interface. ATMI enables task configurations to be controlled via simple C-style structures.

2.3 Dynamic Parallelism in GPUs

Dynamic parallelism (DP) is a feature that is supported by both AMD and NVIDIA GPUs to allow a GPU kernel to launch child GPU kernels at the device without the involvement of CPU. DP can be used to improve the performance of irregular applications by alleviating the workload imbalance and irregularity. For example, Figure 2 shows that dynamic parallelism allows an overloaded parent thread with a number of subtasks larger than *THRESHOLD* (Line 3) to offload their subtasks into child kernels (Line 4). Moreover, in the child kernel, each subtask will be processed in fine-grained parallelism by multiple threads (Line 8), which can better exploit GPU computational resources and memory bandwidth.

2.4 Performance Counters (PCs)

Hardware performance counters (PCs) can help us to perform low-level performance analysis and tuning. In particular, by tracing these PCs, programmers can identify the correlation between programs and their performance. Both AMD and NVIDIA provide profiling tools and APIs to access these performance counters. Table 1 shows an example of performance counters on NVIDIA GPUs.

```
__kernel parent() {
  load this_subtask from subtasks array
  if(size of this_subtask >= THRESHOLD)
    launch child kernel to process this_subtask
  else
    process this_subtask}

__kernel child(this_subtask) {
   process this_subtask by all threads}
```

Figure 2: Example of dynamic parallelism

In this paper, we utilize these performance counters to establish performance models for performance analysis and prediction.

2.5 Statistical and Machine Learning Models

The performance of applications with irregular kernels are affected by runtime factors such as the input data itself. In this paper, we leverage statistical and machine learning models in order to build a general performance model for complex situations. These models allow us to build an accurate model with a small set of measurements, which can significantly reduce the overhead of measurements. In this section, we provide the background of the statistical and machine learning tools that will be used in this paper.

Tree-based Regression Models. Tree-based regression models [13] provide an alternative to classical linear regression. It builds decision trees with training datasets and generates the classification or regression of the individual trees. Random decision forests (Random Forest or RF) [9] is a popular regression-tree model that selects features randomly to avoid the over-fitting issues in decision trees.

Principle Component Analysis (PCA). PCA [13] is a statistical tool to reduce the number of dimensions by converting a large set of correlated variables into a small set of uncorrelated variables (i.e., principal components), where most of the information still remain in the large set. PCA is a technique used to identify the important variables and patterns in a dataset.

Hierarchical Cluster Analysis (HCA). HCA [13] is a statistical and data-mining tool that builds a hierarchy of clusters for cluster analysis. It provides a measure of correlation between sets of observations by using an appropriate metric (e.g., distance matrices) and a linkage criterion that represents the similarity of sets with pairwise distances of observations in the sets.

3 PROBLEMS OF DYNAMIC PARALLELISM

In this section, we characterize the performance issues of existing approaches to dynamic parallelism.

3.1 Experimental Setup

Algorithm Implementations. To identify the performance issues in dynamic parallelism, we choose three typical irregular applications, suffering greatly from load imbalance [10], including *Sparse-Matrix Vector Multiplication* (SpMV), *Single Source Shortest Path* (SSSP), and *Graph Coloring* (GCLR). For each application, we first provide the basic dynamic parallelism implementation that spawns a child kernel per subtask from a thread (Figure 2). And then according to the recent publications [25, 29], we build the existing subtask aggregation approach that consolidates as many as possible subtasks into a GPU kernel to minimize the kernel launch overhead and improve occupancy. As shown in Figure 3, the parent kernel stores all subtasks into a global queue (Line 4), and launches a child kernel for all subtasks, and a subtask is processed by a workgroup for the AMD GPU (or one thread block for the NVIDIA GPU) (Line 11), which was reported to be the best configuration for the graph and sparse linear algebra algorithms.

```
1   __kernel parent(queue) {
2     load this_subtask from subtasks array
3     if(size of this_subtask >= THRESHOLD){
4       push this_subtask into queue  }
5     else{
6       process this_subtask}
7     global synchronization
8     if(globalThreadId == launcher)
9       launch a queue_process kernel to process queue}
10
11  __kernel queue_process(queue) {
12     load this_subtask from queue
13     process this_subtask by a thread block}
```

Figure 3: Example of existing subtask aggregation mechanism

Dataset. Each application has three datasets from the DIMACS challenges [1]: *coPapers*, which has 434,102 nodes and 16,036,720 edges, *kron-logn16*, which has 65,536 nodes and 4,912,142 edges, and *kron-logn20*, which has 1,048,576 nodes and 89,238,804 edges.

Hardware. We evaluate dynamic parallelism on two GPU platforms from AMD and NVIDIA. The AMD platform uses a Vega GPU, and the host consists of two Intel Broadwell CPUs (E5-2637v4). The NVIDIA platform uses a P100 (Pascal) GPU, and the host has two Intel Broadwell CPUs (E5-2680v4).

Compiler. Each application has OpenCL and CUDA versions for AMD and NVIDIA platforms, respectively. OpenCL kernels are compiled and executed with ROCm 1.6 and ATMI v0.3. CUDA kernels are compiled and executed with CUDA 8.0. CPU-side codes are compiled with GCC 4.7.8.

Profilers. To get in-depth performance analysis, we use the profilers provided by NVIDIA and AMD to get the performance counters of GPUs. On the NVIDIA platform, we use *nvprof* from CUDA 8.0. On the AMD platform, we use *CodeXL* of version 2.5.

3.2 Performance Analysis

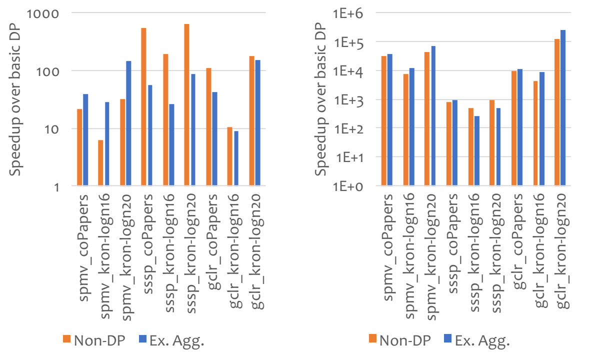
To identify the performance issues in existing approaches, we perform an in-depth performance analysis on our driving applications without dynamic parallelism, implementations with the dynamic parallelism, and the dynamic parallelism with existing subtask aggregation.

Figure 4 illustrates huge performance improvements of using DP with existing subtask aggregation mechanism as well as non-DP solutions over the naïve dynamic parallelism implementation. Moreover, with better workload balance and improved memory access patterns, the existing subtask aggregation can deliver better performance than the implementation without dynamic parallelism (except the SSSP algorithm on the AMD platform). Furthermore, we observe that the speedup of subtask aggregation over the default DP implementation is higher on the NVIDIA platform than on the AMD platform, which is possibly due to the higher per-thread kernel launch overhead on the NVIDIA platform.

Figure 5 shows the normalized execution time of child kernels, including kernel launch time and kernel compute time. We can find that with the subtask aggregation mechanism, the kernel launch

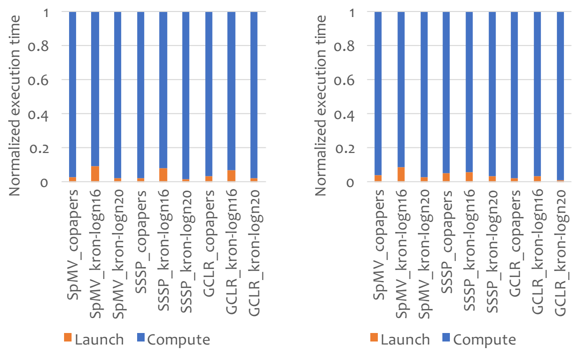
Table 1: Performance counters (NVIDIA Pascal GPU) [8]

Performance Counter	Description
warp_execution_efficiency	Ratio of the average active threads per warp to the maximum number of threads per warp
inst_replay_overhead	Average number of replays for each instruction executed
global_hit_rate	Hit rate for global loads
gld/gst_efficiency	Ratio of requested global memory load/store throughput to required global memory load throughput
gld/gst_throughput	Global memory load/store throughput
gld/gst_requested_throughput	Requested global memory load/store throughput
tex_cache_hit_rate	Texture cache hit rate
l2_read/write_throughput	Memory read/write throughput seen at L2 cache for all write requests
l2_tex_read/write_hit_rate	Hit rate at L2 cache for all read/write requests from texture cache.
issue_slot_utilization	Percentage of issue slots that issued at least one instruction, averaged across all cycles
ldst_issued/executed	Number of issued/executed local, global, shared and texture memory load and store instructions
stall_not_selected	Percentage of stalls occurring because warp was not selected
issued/executed_ipc	Instructions issued/executed per cycle



(a) AMD platform (b) NVIDIA platform

Figure 4: Speedup of the implementations without dynamic parallelism (Non-DP) and the implementations with existing subtask aggregation (Ex. Agg.) over the default dynamic parallelism implementations (Basic DP).



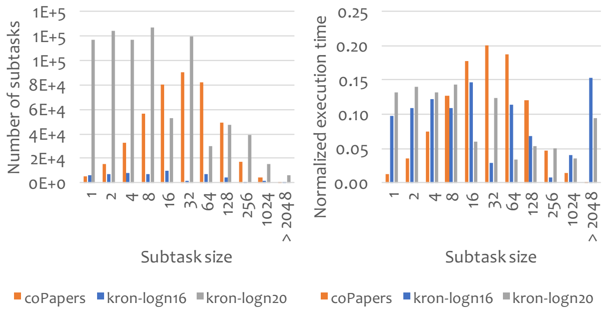
(a) AMD platform (b) NVIDIA platform

Figure 5: Breakdown of the child kernel execution time in the existing subtask aggregation mechanism (Ex. Agg.).

overhead is significantly reduced to be negligible and most of the execution time is spent on the computation of subtasks, especially for large datasets (i.e., *kron-logn20*). Thus, if one wants to improve the overall performance of dynamic parallelism, improving the performance of subtasks is more critical than reducing the launch overhead of child kernels.

Although the subtask aggregation mechanisms can significantly improve the overall performance of dynamic parallelism, with a deeper investigation, we find that there is a major drawback in existing subtask aggregation mechanisms: they treat all subtasks equally by using the "one-size-fits-all" methodology, aggressively aggregate as many subtasks as possible into a single kernel ("N-to-1" approach), and apply the uniform configuration and parallel strategy for all subtasks. However, we have observed there are highly diverse characteristics in subtasks. Figure 6(a) shows that in the SpMV algorithm, the subtask sizes (i.e., corresponding numbers of GPU threads in subtasks) can range from 1 to over 2K; and the distribution of subtask sizes highly depends on the input datasets. We also find although most of the subtasks fall into the range from 1 to 256 in this case, the execution time of large subtasks (i.e., the subtask size > 2048) can take a considerable portion of the total execution time, as shown in Figure 6(b).

As a result, we carry out a simple evaluation to investigate if we can find different performance when we vary the resource usage (i.e., changing GPU workgroup sizes) for different subtask sizes.

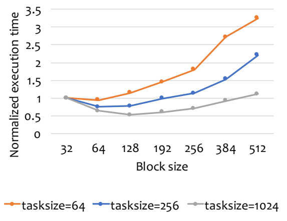


(a) Subtask size (b) Execution time

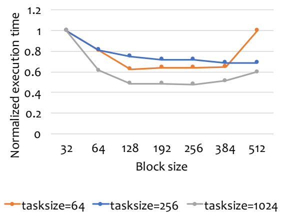
Figure 6: The distribution of subtask size and execution time of the SpMV algorithm. The execution time of each subtask size is normalized to the total execution time.

Figure 7 shows that for the subtasks of size 64, 256, and 1024, their performances have obviously affected by the workgroup size;

and the optimal workgroup size is variable with the subtask size and algorithm. The major reason is that when we change the workgroup size for a given algorithm, each thread has different workloads and uses different hardware resources (e.g., GPU registers, local memory, leading to changes in parallelism, occupancy, and resource utilization). As a consequence, the "one-size-fits-all" approach in existing approaches may result in resource underutilization. A more intelligent subtask aggregation strategy is needed.



(a) SpMV



(b) SSSP

Figure 7: Performance of SpMV and SSSP algorithms with different block size. The execution time is normalized to that of block size = 32.

4 ADAPTIVE SUBTASK AGGREGATION

To obtain the optimal task aggregation strategy for dynamic parallelism, we propose a task aggregation modeling and task scheduling tool that uses statistical analysis and machine learning techniques to establish performance models based on a collection of performance counters with sampling data.

Figure 8 shows the high-level depiction of our tool, which consists of four phases: 1) *performance measurement* phase, which collects performance counter data from all performance counters via profilers with different input datasets and parameters; 2) *performance modeling* phase, which establishes a performance analysis model (i.e., determining most important performance counters and subtask characteristics); 3) *performance prediction* phase, which uses the previously identified important performance counters and characteristics to build a performance prediction model; 4) *aggregation generation* phase, which generates the optimal subtask aggregation strategy based on the performance model by considering subtask performance gain and loss, aggregation overhead, and kernel launch overhead. Below we will discuss each phase in details.

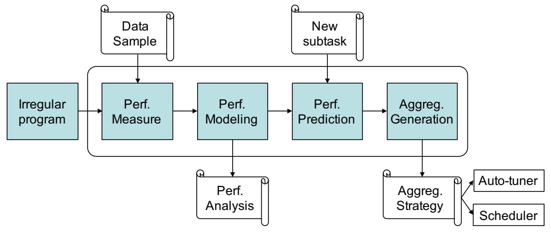


Figure 8: Architecture of our adaptive subtask aggregation

4.1 Performance Measurement

The performance measurement phase collects performance counter data of the irregular program on the target architecture. Since the collection of performance counter data can significantly affect the accuracy of the performance models in the later stages, we carry out the performance measurement by running the subtasks with varying parameters, including different datasets, subtask sizes, and runtime configurations (i.e., workgroup size and the number of workgroups). During the performance measurement, our tool collects performance counter data, and measures the execution time as the response variable. Performance counter data are collected from all performance counters using corresponding performance profilers - *CodeXL* and *nvprof* for AMD and NVIDIA platforms, respectively.

The size and selection of sample data are critical for the accuracy of the performance model. Though more sample data can improve the accuracy of the performance model, over-saturated sample data will significantly increase the data collection time and performance modeling overhead. Moreover, to avoid selection bias, which makes the model is non-representative for new subtasks with unseen characteristics, the data selection should have proper randomization. According to experiments on our implementations, 200 samples with randomly selected different input parameters and configurations are sufficient to build accurate performance models for predicting the optimal aggregation strategy. As a configurable parameter, the number of samples can also be set by users.

4.2 Performance Modeling

In the performance modeling phase to identify the most important performance factors, we utilize statistical and machine learning tools, including Principal Components Analysis (PCA), Random Forest Regression (RF) and Hierarchical Cluster Analysis (HCA).

4.2.1 Principal Components Analysis (PCA). We first perform PCA analysis on performance counter data, which can help us to identify important performance counters that contribute most to the variance, and also can help us to determine the correlation between these performance counters. Based the importance and correlation, we can reduce the number of performance counters for the later performance modeling to reduce the risk of over-fitting. In this paper, we identify first few important performance counters (< 10) from the top principal components as important variables.

4.2.2 Random Forest Regression. After PCA analysis, we apply the Random Forest (RF) model on the performance counter data, and obtain the relative variable importance of RF, which can reveal the influence of a variable to the response variable (i.e., execution time). Through identifying the most important variables (i.e., performance counters), we can determine the performance counters that are strongly correlated to the execution time, which give us hints of the characteristics and performance bottlenecks of subtasks.

4.2.3 Hierarchical Cluster Analysis (HCA). After the Random Forest, we use Hierarchical Clustering Analysis (HCA) to help us get insights of the important performance counters determined by the Random Forest, which can give us hints about the characteristics and performance bottlenecks of subtasks.

4.2.4 Result Analysis. In this section, we offer examples of performance analysis results of performance modeling phase.

SpMV. Figure 9 shows the result of performance modeling of the SpMV algorithm. Based on the PCA results (Figure 9(a)), we can identify the most variable performance counters from the top four principle components - PC1, PC2, PC3 and PC4, which account for most of the variance in the performance counter data. The most variable performance counters are *global_hit_rate*, *tex_cache_hit_rate*, *gld_throughput*, *achieved_occupancy*, *ldst_issued*, and *ldst_executed*. After identifying the most variable performance counters, the Random Forest will be applied to the performance counter data with execution time as response variable to determine the most important performance counters relevant to performance. Figure 9(b) shows the *inst_issued* and *inst_replay_overhead* are the two most important performance counters for the SpMV algorithm. Then, we can turn to HCA to get more insights. We can observe that the most relevant performance counter (i.e., *inst_issued*) has strong correlation to *inst_issued* and *ldst_executed*. And the second important performance counter -*inst_replay_overhead* has strong correlation to *global_hit_rate*, *tex_cache_hit_rate* and *l2_tex_write_hit_rate*. It indicates that data locality and the amount of workload have significant impacts on the performance of SpMV.

SSSP. From the results for SSSP (Figure 10(a)), we can observe that the SSSP algorithm has highly similar PCA results as the SpMV algorithm. However, Figure 10(b) shows the most important variable for the time prediction is *dram_write_throughput*. From Figure 10(c), we observe that *dram_write_throughput* is strongly connected to *inst_replay_overhead*, *gst_efficiency*, *l2_tex_write_hit_rate* and *gst_throughput*. It can give us a hint that the performance of SSSP is highly relevant to the memory write performance.

Graph Coloring. Figure 11 shows the performance modeling result of the Graph Coloring algorithm. Similar with the SSSP algorithm, *dram_write_throughput* is the most important performance counters for performance prediction. Based on the result of HCA (Figure 11(c)), there are high correlation among *dram_write_throughput*, *l2_tex_read_hit_rate*, *gld_efficiency*, and *dram_read_throughput*, which indicate that the performance of graph coloring is highly relevant to the memory read performance.

4.3 Performance Prediction

Based on the performance modeling phase, we can establish a prediction model to estimate the performance of new subtasks.

4.3.1 Prediction Model with Random Forest. Our general idea for the performance prediction is (1) building dedicated prediction models for each top important performance counters (≤ 5) based on workgroups size, subtask size, and the number of subtasks; (2) using the top important performance counters to retrain the prediction model for the execution time; (3) and merging the two sets of predict models to predict the execution time for new subtasks with the given subtask size and the number of tasks. With this performance prediction model, we can predict the optimal workgroup size for new subtasks. Then, we can perform the initial subtask aggregation that groups subtasks with the same workgroup size together and set the optimal workgroup size for each group. However, to achieve the optimal overall performance, we need a more sophisticated subtask aggregation strategy, which will be discussed in the following Section 4.4.

4.3.2 Result Analysis. Figure 12 shows an example of performance prediction model for SpMV subtasks with task size = 128, the number of tasks = 64, and varied workgroup size. To verify the accuracy of our model, we randomly select 80% of data as training data and use the rest 20% of data as evaluation data. As Figure 12(a) shows, we first predict the top five important performance counters. Then, as Figure 12(b) illustrates, we use the predicted value of the top five performance counters to estimate the performance of the given SpMV subtasks with varying workgroup size.

Figure 13 shows the prediction results of SSSP and GCLR algorithms. In general, we get produce highly accurate performance prediction for all algorithms, which can determine the optimal block size for given tasks size and number of tasks. In this example, the optimal block size for the SpMV algorithm is 128 (Figure 12(b)), while the optimal block size for SSSP and GCLR algorithms is 256 (Figure 13).

4.4 Aggregation Generation

With the performance prediction model, we can easily determine the optimal configurations (i.e., block size) for new subtasks by searching the configuration space. However, generating the optimal aggregation strategy is non-trivial and has the following challenges.

First, despite that kernel launch time has been continually reduced in the latest GPU micro-architecture and runtime, the kernel launch is not free. Second, the aggregation will reduce kernel launch overhead, but aggregating subtasks with different configurations will result in performance loss by applying non-optimal configuration on subtasks. Third, subtask aggregation also introduces aggregation overhead (e.g., migrating subtasks into the same subtask group). Therefore, we need to balance the kernel launch overhead, aggregation overhead, and subtask performance.

To achieve the optimal overall performance, we build a model that firstly identifies the optimal performance for each subtask (Equation 1), and then searches the subtasks, which require the identical or similar configuration (i.e., block size), into a kernel, and uses a subtask aggregation model to estimate the optimal performance after merging subtasks, and then determines if we need to merge the two subtask groups. We estimate the optimal task time by applying the configuration across each other subtask to choose the optimal one (Equation 2) for both, and then determine merge or not by considering the kernel launch overhead and aggregation overhead (Equation 3). Note that the existing aggregation method only considers reducing kernel launch overhead rather than the performance loss of applying the non-optimal configuration.

$$\begin{aligned} conf_A, time_A &= config\_search(task_A) \\ conf_B, time_B &= config\_search(task_B) \end{aligned} \tag{1}$$

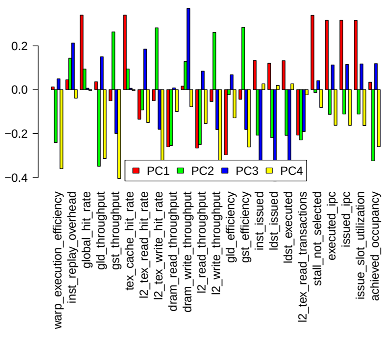
$$time_{child} = min \begin{cases} predict(conf_A, t_B) + time_A \\ predict(conf_B, t_A) + time_B \end{cases} \tag{2}$$

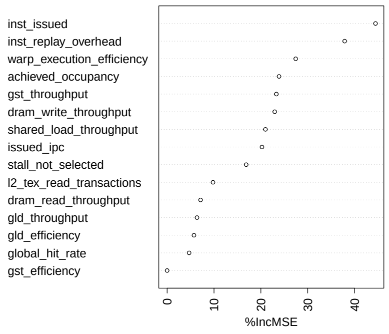
$$time_{overall} = min \begin{cases} time_A + time_B \\ time_{child} - time_{kl} + time_{agg} \end{cases} \tag{3}$$

5 PERFORMANCE EVALUATION

Here, we evaluate the effectiveness of our optimal aggregation model on AMD and NVIDIA platforms, and compare it with the performance of existing aggregation techniques. The existing subtask aggregation (Ex. Agg.) aggressively consolidates as many possible subtasks into a single kernel. On the other hand, our optimal subtask aggregation (Opt. Agg.) technique strategically aggregates subtasks based on our subtask aggregation model.

As the performance measurement, performance modeling and performance prediction phases are offline, they just need to run once offline with sampling data for an application. Therefore, in the evaluation, we do not include the execution time of these phases,


(a) Factor loadings



(b) Variable importance

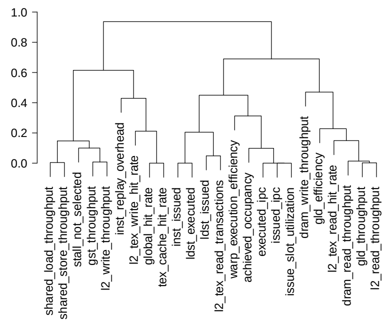
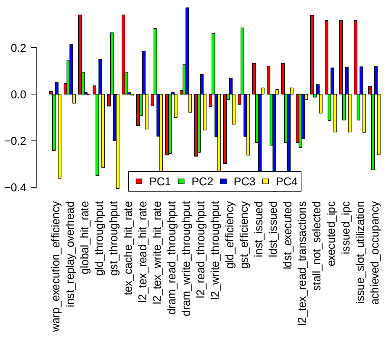
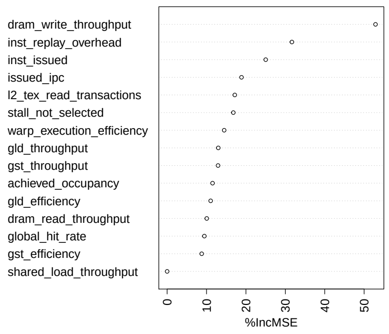

(c) Clustering

Figure 9: The result of the performance modeling of the SpMV algorithm.


(a) Factor loadings



(b) Variable importance

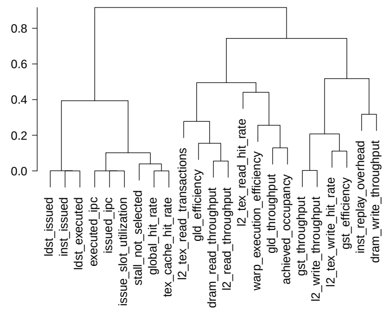
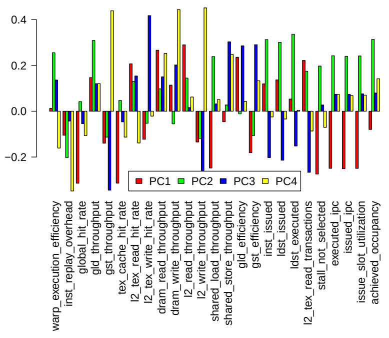
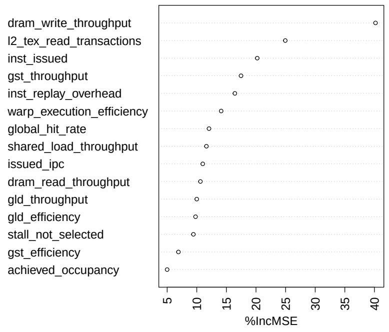

(c) Clustering

Figure 10: The result of the performance modeling of the SSSP algorithm.


(a) Factor loadings



(b) Variable importance

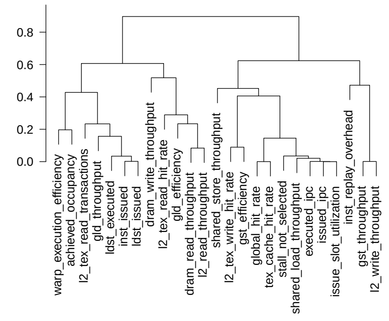

(c) Clustering

Figure 11: The result of the performance modeling of the Graph Coloring (GCLR) algorithm.

only include the execution time of the aggregation generation phase, which needs to be performed at runtime.

5.1 Performance Comparison

Figure 14 shows the performance comparison between the existing aggregation and optimal subtask aggregation. The optimal aggregation strategy can achieve up to a 1.8-fold speedup over the existing aggregation on the NVIDIA platform, and a 1.5-fold speedup on the AMD platform. For dataset *kron-logn20*, which has higher subtask size diversity, we can achieve higher performance improvement.

5.2 Performance Profiling

With in-depth profiling, as Figure 15 shown, the implementation with the optimal aggregation improves *warp_execution_efficiency* and *global_hit_rate*, which indicate better load balance, and less irregular memory accesses, respectively. Furthermore, we observe the noticeable improvement in *achieved_occupancy*, which indicates improved resource utilization that is crucial for the performance of dynamic parallelism.

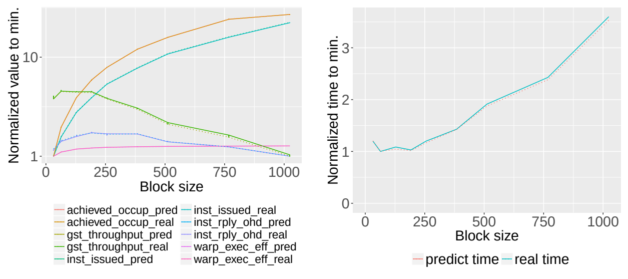

(a) Performance counter prediction (b) Performance prediction

Figure 12: The result of the performance prediction of the SpMV algorithm with *kron-logn16* dataset, task size = 128 and number of tasks = 64 (normalized to minimal time)

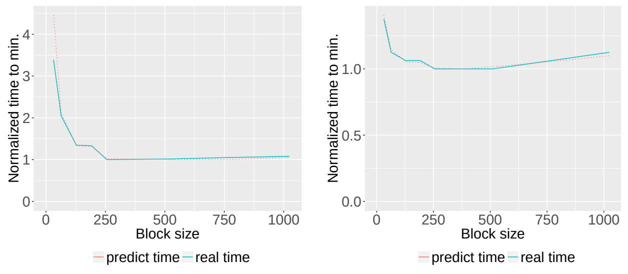

(a) SSSP (b) GCRL

Figure 13: The result of the performance prediction of SSSP and GCLR algorithms *kron-logn16* dataset, task size = 128 and number of tasks = 64 (normalized to minimal time)

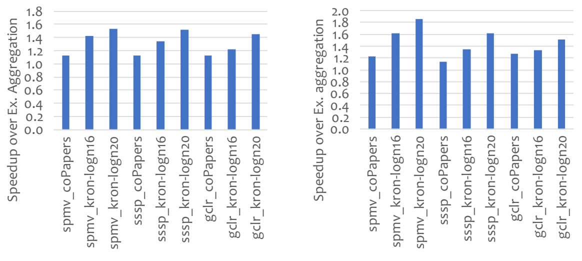
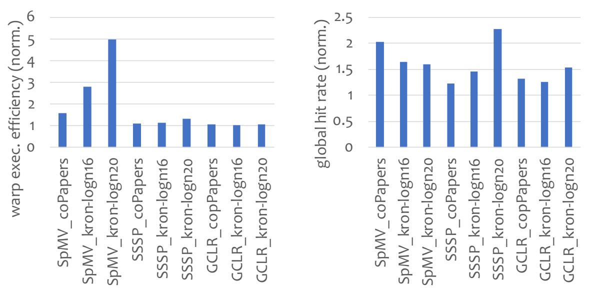

(a) AMD platform (b) NVIDIA platform

Figure 14: Speedup of the optimal subtask aggregation over the existing subtask aggregation (Ex. Agg.)

6 RELATED WORK

Currently, many applications in well-established and emerging fields (e.g., graph algorithms [24, 31], sparse linear algebra [33, 34], security analysis, and bioinformatics algorithms [32, 35, 36]) exhibit increasing irregularities in memory access, control flow, and communication and I/O patterns. It is very challenging to map these irregular applications on a GPU.

Dynamic Parallelism. Prior work on dynamic parallelism for GPUs mainly focuses on kernel launch overhead. Wang *et al.* [27] characterize the benefits and overheads of dynamic parallelism for irregular applications. They then propose dynamic thread block launch (DTBL) [25], a hardware-based subtask aggregation that buffers subtasks in aggregation tables. To further improve the efficiency of dynamic parallelism, this work is enhanced by a locality-aware scheduler [26]. Orr *et al.* [19] also provide a subtask aggregation scheme in hardware for fine-grained tasks. All these hardware-based approaches require hardware modification, while our software-based approach improves the efficiency of dynamic parallelism on current GPU architectures.


(a) Warp execution efficiency (b) Global hit rate

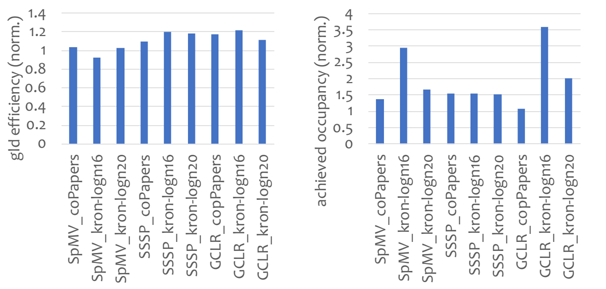

(c) Global load efficiency (d) Achieved occupancy

Figure 15: Normalized profiling numbers of optimal aggregation (Opt. Agg.) over existing aggregation (Ex. Agg.) on the NVIDIA platform

Besides hardware-based subtask aggregation, multiple compiler-based approaches use subtask aggregation to reduce the number of kernel launches. Yang *et al.* propose CUDA-NP [30], a compiler approach for exploring nested parallelism via using slave threads in a thread block to process subtasks. Chen *et al.* [11] propose "Free Launch" which reuses the parent threads to process child tasks. Wang *et al.* [23] present *kernel fusion* to achieve high utilization. Wu *et al.* [29] propose a subtask aggregation of child kernels at three different granularities — warp, block, and kernel. Similarly, Hajj [12] aggregates kernels at the same three granularities and overlaps child kernel execution with parent kernel via dispatching child kernel prior to child tasks ready. Paravecino proposes a framework to exploit the nest parallelism and concurrent kernel execution via launching child kernels [20]. However, all these software approaches focus on reducing kernel launch overhead regardless of child kernel performance.

Performance Modeling on GPUs. Many approaches to GPU performance modeling utilize machine learning [14–16, 18, 21, 22, 28, 37, 38].Zhang *et al.* [37] propose a statistical approach to identify the relationship between characteristics of a kernel on a GPU and its performance and power consumption. Souley *et al.* [18] propose a statistical model based on Random Forest (RF) to characterize and predict the performance of GPU kernels. Rogers *et al.* [21] characterize the effect of the warp-size on NVIDIA GPUs. Yao *et al.* [38] characterize some performance factors for NVIDIA GPUs including work-group sizes. Stargazer [15] is an automated GPU performance framework based on stepwise regression modeling. Eiger [16] is an automated statistical methodology for modeling program behavior on different architectures. Though considerable attention has been focused on performance models to provide performance analysis and prediction on GPU architectures, *none* of them address the subtasks of dynamic parallelism in GPUs, which are tiny, irregular, and many in numbers.

7 CONCLUSION

While dynamic parallelism can potentially improve the performance of irregular applications, existing approaches deliver sub-optimal performance due to high kernel launch overhead and low subtask occupancy. Our in-depth performance characterization of existing approaches, which treats all subtasks equally and use uniform configurations, shows that GPU under-utilization can occur due to variable characteristics between subtasks. To overcome these challenges, we propose a subtask aggregation and scheduling tool that 1) establishes a set of performance models for performance analysis and prediction of subtasks based on statistical and machine-learning techniques and 2) generates the optimal subtask aggregation strategy by considering the subtask performance, kernel launch, and aggregation overhead. Experimental results show that the optimal subtask aggregation strategy generated by our tool can achieve up to a 1.8-fold speedup over the existing subtask aggregation.

8 ACKNOWLEDGMENTS

This research was supported in part by the NSF XPS program via CCF-1337131 and the Synergistic Environments for Experimental Computing (SEEC) Center. Jing Zhang performed the work at AMD as an internship and currently is an AMD employee. Hao Wang performed the work at Virginia Tech and is currently a Research Scientist at Ohio State University.

REFERENCES

[1] DIMACS Implementation Challenges. https://www.cc.gatech.edu/dimacs10. (2012).

[2] The OpenCL Specification v2.0. https://www.khronos.org/registry/OpenCL/specs/opencl-2.0.pdf. (2015).

[3] NVIDIA Tesla P100 Whitepaper. https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf. (2016).

[4] Asynchronous Task and Memory Interface (ATMI). https://github.com/RadeonOpenCompute/atmi. (2017).

[5] CUDA C Programming Guide. (2017).

[6] Radeon's Next-generation Vega architecture. https://radeon.com/_downloads/vega-whitepaper-11.6.17.pdf. (2017).

[7] ROCm - Open Source Platform for HPC and Ultrascale GPU Computing. https://rocm.github.io/install.html. (2017).

[8] Profiler User's Guide. http://docs.nvidia.com/cuda/pdf/CUDA_Profiler_Users_Guide.pdf. (2018).

[9] L. Breiman. Random Forests. *Machine Learning* 45, 1 (2001), 5–32.

[10] M. Burtscher, R. Nasre, and K. Pingali. A Quantitative Study of Irregular Programs on GPUs. In *2012 IEEE Int. Symp. on Workload Characterization.* 141–151.

[11] G. Chen and X. Shen. Free Launch: Optimizing GPU Dynamic Kernel Launches Through Thread Reuse. *2015 Int. Symp. on Microarchitecture*, 407–419.

[12] I. Hajj, J. Gomez-Luna, C. Li, L. W. Chang, D. Milojicic, and W. M. Hwu. KLAP: Kernel Launch Aggregation and Promotion for Optimizing Dynamic Parallelism. In *2016 Int. Symp. on Microarchitecture.*

[13] T. Hastie, R. Tibshirani, and J. Friedman. *The elements of statistical learning: data mining, inference and prediction.* Springer.

[14] K. Hou, W. c. Feng, and S. Che. Auto-Tuning Strategies for Parallelizing Sparse Matrix-Vector (SpMV) Multiplication on Multi- and Many-Core Processors. In *2017 IEEE Int. Parallel Distrib. Processing Symp. Workshop.*

[15] W. Jia, K. A. Shaw, and M. Martonosi. Stargazer: Automated regression-based GPU design space exploration. In *2012 IEEE Int. Symp. on Performance Analysis of Systems Software.* 2–13.

[16] A. Kerr, E. Anger, G. Hendry, and S. Yalamanchili. Eiger: A framework for the automated synthesis of statistical performance models. In *2012 Int. Conf. on High Performance Computing.* 1–6.

[17] D. Li, H. Wu, and M. Becchi. Nested Parallelism on GPU: Exploring Parallelization Templates for Irregular Loops and Recursive Computations. In *2015 Int. Conf. on Parallel Processing.* 979–988.

[18] S. Madougou, A. L. Varbanescu, C. D. Laat, and R. V. Nieuwpoort. A Tool for Bottleneck Analysis and Performance Prediction for GPU-Accelerated Applications. In *2016 IEEE Int. Parallel Distrib. Processing Symp. Workshops.* 641–652.

[19] M. S. Orr, B. M. Beckmann, S. K. Reinhardt, and D. A. Wood. Fine-Grain Task Aggregation and Coordination on GPUs. In *2014 Int. Symp. on Computer Architecture.* 181–192.

[20] F. N. Paravecino. *Characterization and exploitation of nested parallelism and concurrent kernel execution to accelerate high performance applications.* Ph.D. Dissertation. Massachusetts : Northeastern University, Boston.

[21] T. G. Rogers, M. O'Connor, and T. M. Aamodt. Cache-Conscious Wavefront Scheduling. In *2012 IEEE/ACM Int. Symp. on Microarchitecture.* IEEE Computer Society, Washington, DC, USA, 72–83.

[22] Y. Ukidave, X. Li, and D. Kaeli. Mystic: Predictive Scheduling for GPU Based Cloud Servers Using Machine Learning. In *2016 IEEE Int. Parallel Distrib. Processing Symp.* 353–362.

[23] G. Wang, Y. Song Lin, and W. Yi. Kernel fusion: An effective method for better power efficiency on multithreaded GPU. In *2010 IEEE/ACM Int. Conf. on Green Computing and Communications.* 344–350.

[24] H. Wang, W. Liu, K. Hou, and W. c. Feng. Parallel Transposition of Sparse Data Structures. In *2016 Int. Conf. on Supercomputing.* Article 33, 13 pages.

[25] J. Wang, N. Rubin, A. Sidelnik, and S. Yalamanchili. Dynamic Thread Block Launch: A Lightweight Execution Mechanism to Support Irregular Applications on GPUs. In *2015 Int. Symp. on Computer Architecture.* 528–540.

[26] J. Wang, N. Rubin, A. Sidelnik, and S. Yalamanchili. LaPerm: Locality Aware Scheduler for Dynamic Parallelism on GPUs. In *2016 Int. Symp. on Computer Architecture.* 583–595.

[27] J. Wang and S. Yalamanchili. Characterization and Analysis of Dynamic Parallelism in Unstructured GPU Applications. In *2014 IEEE Int. Symp. on Workload Characterization.* 51–60.

[28] G. Wu, J. L. Greathouse, A. Lyashevsky, N. Jayasena, and D. Chiou. GPGPU performance and power estimation using machine learning. In *2015 IEEE Int. Symp. on High Performance Computer Architecture.* 564–576.

[29] H. Wu, D. Li, and M. Becchi. Compiler-Assisted Workload Consolidation for Efficient Dynamic Parallelism on GPU. In *2016 IEEE Int. Parallel Distrib. Processing Symp.* 534–543.

[30] Y. Yang, C. Li, and H. Zhou. CUDA-NP: Realizing Nested Thread-Level Parallelism in GPGPU Applications. *Journal of Computer Science and Technology* 30, 1 (2015), 3–19.

[31] J. Yin, J. Wang, W. c. Feng, X. Zhang, and J. Zhang. SLAM: Scalable Locality-aware Middleware for I/O in Scientific Analysis and Visualization. In *2014 Int. Symp. on High-performance Parallel and Distrib. Computing.* 257–260.

[32] J. Yin, J. Zhang, J. Wang, and W. c. Feng. SDAFT: A novel scalable data access framework for parallel BLAST. *Parallel Comput.* 40, 10 (2014), 697 – 709.

[33] X. Yu, H. Wang, W. c. Feng, H. Gong, and G. Cao. An Enhanced Image Reconstruction Tool for Computed Tomography on GPUs. In *2017 Computing Frontiers Conference.* 97–106.

[34] X. Yu, H. Wang, W. c. Feng, H. Gong, and G. Cao. GPU-Based Iterative Medical CT Image Reconstructions. *Journal of Signal Processing Systems* (2018).

[35] D. Zhang, H. Wang, K. Hou, J. Zhang, and W. c. Feng. pDindel: Accelerating indel detection on a multicore CPU architecture with SIMD. In *2015 IEEE Int. Conf. on Computational Advances in Bio and Medical Sciences.* 1–6.

[36] J. Zhang, H. Wang, and W. c. Feng. cuBLASTP: Fine-Grained Parallelization of Protein Sequence Search on CPU+GPU. *IEEE/ACM Trans. Comput. Biol. Bioinf.* 14, 4 (2017), 830–843.

[37] Y. Zhang, Y. Hu, B. Li, and L. Peng. Performance and Power Analysis of ATI GPU: A Statistical Approach. In *2011 IEEE Int. Conf. on Networking, Architecture, and Storage.* 149–158.

[38] Y. Zhang and J. D. Owens. A quantitative performance analysis model for GPU architectures. In *2011 IEEE Int. Symp. on High Performance Computer Architecture.* 382–393.