# An Accurate GPU Performance Model for Effective Control Flow Divergence Optimization

Yun Liang, *Member, IEEE*, Muhammad Teguh Satria, Kyle Rupnow, and Deming Chen, *Member, IEEE*

***Abstract*—Graphic processing units (GPUs) are composed of a group of single-instruction multiple data (SIMD) streaming multiprocessors (SMs). GPUs are able to efficiently execute highly data parallel tasks through SIMD execution on the SMs. However, if those threads take diverging control paths, all divergent paths are executed serially. In the worst case, every thread takes a different control path and the highly parallel architecture is used serially by each thread. This control flow divergence problem is well known in GPU development; code transformation, memory access redirection, and data layout reorganization are commonly used to reduce the impact of divergence. These techniques attempt to eliminate divergence by grouping together threads or data to ensure identical behavior. However, prior efforts using these techniques do not model the performance impact of any particular divergence or consider that complete elimination of divergence may not be possible. Thus, we perform analysis of the performance impact of divergence and potential thread regrouping algorithms that eliminate divergence or minimize the impact of remaining divergence. Finally, we develop a divergence optimization framework that analyzes and transforms the kernel at compile-time and regroups the threads at runtime. For the compute-bound applications, our proposed metrics achieve performance estimation accuracy within 6.2% of measured performance. Using these metrics, we develop thread regrouping algorithms, which consider the impact of divergence, and speed up these applications by 2.2× on average on NVIDIA GTX480.**

***Index Terms*—Control flow divergence, CUDA, GPGPU, performance metric, thread regrouping.**

## I. Introduction

GRAPHICS processing units (GPUs) have been widely used for computational acceleration for embedded systems. A highly parallel architecture paired with a simplified programming model allows easy acceleration of data-parallel applications. Indeed, GPUs benefit a variety of embedded applications including electronics design automation, imaging, *IP routing, audio, aerospace, military, and medical applications [1]–[3]. Recent years have also seen a rapid adoption of GPUs in mobile devices such as smartphones. Mobile devices typically use system-on-a-chip (SoC) that integrates GPUs with CPUs, memory controllers, and other application-specific accelerators. The major SoCs with integrated GPUs available in the market include NVIDIA Tegra series with low power GPU, Qualcomm's Snapdragon series with Adreno GPU, and Samsung's Exynos series with ARM Mali GPU.

In hardware, GPUs are composed hierarchically. A GPU is composed of multiple streaming multiprocessors (SMs), each of which has many streaming processors (SPs) used in a single-instruction multiple data (SIMD) execution style.[1] An SM coordinates the SIMD style execution of many threads, divided into groups of 32 threads called warps. Threads within a warp start together at the same program address; each thread has its own program counter (PC) and register state. If all the threads in a warp execute the same instruction (same PC value), they may all execute in parallel. However, threads may follow different paths at conditional branches. When the threads within a warp diverge, the SM must coordinate serial execution of threads that follow different paths, and then later reconverge these threads. In the worst case, each thread in a warp takes a different control path; thus, only one thread of the warp is active at a time, and there is no actual parallel execution. Control flow divergence can significantly affect the performance of GPU applications. For example, Baghsorkhi *et al.* [4] demonstrated a prefix-scan benchmark with 33% of total execution time spent in control flow divergences, and some of the benchmarks used within this paper waste more execution latency on control flow divergence.

Although control flow divergence is a well-known and well-studied problem, it cannot necessarily be entirely eliminated. Complex programs often have several divergence points at different stages of execution; in practice, it is not possible to entirely eliminate divergence through regrouping of threads or data inputs. Prior attempts to eliminate control flow divergence concentrated on thread regrouping techniques such as dynamic warp formation [5] or subdivision [6], memory access indirection, or input data transformations [7], [8]. Although each of these techniques is effective to reduce specific instances of

Manuscript received June 15, 2015; revised September 17, 2015; accepted October 17, 2015. Date of publication November 16, 2015; date of current version June 16, 2016. This work was supported in part by the National Natural Science Foundation of China under Grant 61300005, in part by the Human-Centered Cyber-Physical Systems Programme at the Advanced Digital Sciences Center from Singapore's Agency for Science, Technology and Research (A*STAR) and CFAR, one of six centers of STARnet, and in part by the Semiconductor Research Corporation program sponsored by MARCO and DARPA. This paper was recommended by Associate Editor P. R. Panda. *(Corresponding author: Yun Liang.)*

Y. Liang is with the Center for Energy-Efficient Computing and Applications, School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China, and also with the Collaborative Innovation Center of High Performance Computing, National University of Defense Technology, Changsha 410073, China (e-mail: ericlyun@pku.edu.cn).

M. T. Satria and K. Rupnow are with the Advanced Digital Science Center, Illinois at Singapore, Singapore.

D. Chen is with the University of Illinois at Urbana–Champaign, Champaign, IL 61801 USA.

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2015.2501303

[1]In this paper, we use NVIDIA GPUs and terminology, but the concepts also apply to other GPGPUs.

control flow divergence, no prior technique models the performance impact of the transformations or prioritizes which instances of divergence are most important to eliminate. Prior elimination techniques simply assume that all instances of divergence are equally important and thus a reduction in percentage of divergent warps [7] or percentage of divergent branches [9] serves as their metrics for estimating improvement. However, these metrics are insufficient for estimating performance impact of divergence; divergences last for different numbers of instructions, instruction latency varies, and thread block scheduling can impact overlapping of threads' execution, thus each divergence will have different importance to kernel execution latency.

Performance models for GPU kernels are effective techniques for analyzing kernel behavior and predicting performance on a particular platform. Prior performance models have been used to estimate memory warp parallelism [10] and overall GPU performance [4]. However, these models either ignore divergence entirely, or use statistical modeling rather than thread-specific modeling, which is insufficient to guide thread regrouping for performance improvement. For these reasons, we develop an accurate performance model for control flow divergence that adds per-thread execution information in order to accurately estimate overall application performance of candidate thread regrouping algorithms.

In this paper, we first develop performance metrics that accurately predict the performance impact of control flow divergence for compute-bound GPU kernels. Specifically, our performance metrics combine per-thread basic block vector (BBV), instruction latencies, basic block instruction counts, number of simultaneously supported thread blocks, and thread block scheduling policy. BBV captures the execution footprint of threads. More clearly, BBV combines the list of all the basic blocks of the program with the count of how many times each basic block is executed. Then, we develop several thread regrouping algorithms to eliminate control flow divergence based on the BBVs. Finally, we propose a divergence optimization framework that puts the models and algorithms together. By comparing the relative improvement of the actual performance and metric, we demonstrate that our proposed metrics correlate well with the performance on NVIDIA GTX480. Our divergence optimization demonstrates performance speedup for a range of compute-bound, control flow diverging applications. This paper advanced the state of the art of control flow divergence modeling and optimization for GPUs with the following.

1) We develop novel control flow divergence metrics based on the BBV, warp and thread block performance models, to predict control flow divergence impact on kernel performance.
2) We develop thread regrouping algorithms to significantly reduce control flow divergence based on our performance metrics.
3) We develop a divergence optimization framework that performs kernel analysis and transformations at compile-time and enable the thread regrouping at runtime.

We demonstrate that for compute-bound applications, our metrics achieve performance estimation accuracy within 6.2% of the measured performance on NVIDIA GTX480 GPU. Our divergence optimization speedups these kernel execution by 2.2× on average (up to 4.7×) on NVIDIA GTX480.

The rest of this paper is organized as follows. Section II introduces the architecture of GPU and CUDA programming model. Section III presents background information on the control flow divergence problem and software elimination mechanism. Section IV introduces our proposed performance metrics for estimating the performance impact of control flow divergence. Section V presents thread regrouping algorithms and Section VI describes the control flow divergence optimization framework. Section VII demonstrates the experimental results. Finally, Section VIII discusses the related work and Section IX concludes this paper and suggests future work.

## II. GPU Architecture and Programming Model

In this section, we begin by introducing the key architectural features of GPU platforms necessary for control flow divergence performance modeling, and then describe the programming model.

### A. NVIDIA GPU Architecture

In NVIDIA GPUs, each SM coordinates the SIMD style execution of all of the SPs on it. The SPs are deeply pipelined with heterogenous functional unit latencies (depending on operation complexity) and use a static in-order pipeline without data bypassing to minimize the size and complexity of each SP. To maintain high throughput despite program data dependencies, each SM can support many simultaneous threads that share access to the deeply pipelined SPs. Tasks such as instruction fetch and decode, selection of the next instruction(s) to execute, and coordination of branch reconvergence are handled at the SM level. An SM has a variety of architectural parameters that must be honored by the kernel implementation. Some of these parameters are fixed, such as the limitation of 32 threads per warp and the size of scheduling unit. Other parameters such as threads per thread-block, thread-blocks per SM, and shared memory per SM vary by architectures but have fixed limits for any particular architecture.

### B. NVIDIA CUDA Programming Model

The CUDA programming model is closely tied to the underlying architecture; a GPU kernel is composed of many threads hierarchically organized into groups that will execute together in the GPU hardware. At the lowest level of the hierarchy, threads are grouped into warps, sets of 32 threads. Threads within a warp execute in SIMD style, only a single instruction may be fetched, decoded, and executed within a particular cycle, so divergent threads will be serialized.

At the next level of hierarchy is a thread block; a thread block is composed of one or more warps, with an architectural limit on warps per thread block. A thread block is the allocation unit for GPU execution; thread blocks are assigned to execute on one and only one SM. A single SM can typically support execution of multiple thread blocks simultaneously, where the exact number of thread blocks supported is limited by the resource demand per thread block, resource budget on an SM, and architecture limits. There may be more than one

```
Input data: 1   12   1   12   1   12   1   12
       tid : 0   1    2   3    4   5    6   7
```

(a)

```
if (input[tid] > 10){
        ...

} else {


        ...
}
```

(b)

```
for ( int  i = 0; i < input[tid]; i++){
        ...
        ...
}
```

Fig. 1. Program with divergences. (a) If statement. (b) Loop statement.

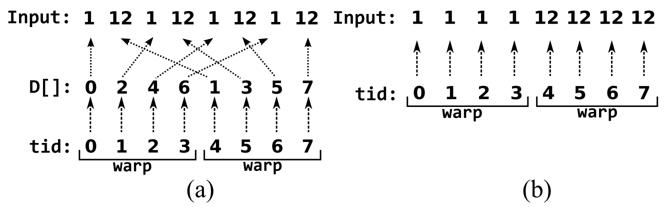


Fig. 2. Thread-data remapping. (a) Reference redirection. (b) Data layout transformation. Warp size is assumed to be 4.

thread blocks defined by a kernel than can be supported for simultaneous execution. The thread blocks will be queued and assigned to SMs using a dynamic round-robin assignment to assign a waiting thread-block to the first available SM. In the case that thread blocks have variable execution latency due to control flow divergence, the ordering of thread blocks can have an impact on load balancing among the SMs and thus on total kernel execution latency.

## III. GPU Control Flow Divergence

We will now discuss the control flow divergence problem and existing elimination mechanisms. Within a warp, threads execute together in an SIMD style. However, every thread has a unique identifier, PC, and register state; thus threads may follow different execution paths at conditional branches. When threads within a warp take different (divergent) paths, each independent execution path will execute serially, and will reconverge at a later convergence point [11]. In particular, branch reconvergence is handled by a combination of special instructions inserted by the compiler and a hardware waiting mechanism. During compilation, nvcc compiler detects cases that may diverge. Through static analysis, nvcc determines the reconvergence point—the PC that all threads will eventually reach regardless of execution path.[2] The compiler inserts a special instruction to identify this PC value just before a divergence point. Then, a hardware mechanism implements a barrier at that PC value—no thread will continue until all have reached that PC value, thus guaranteeing reconvergence [11].

GPU kernels with control flow statements may diverge either due to the input data, or due to the systematic divergence based on the thread IDs. Fig. 1 illustrates branching scenarios based on the input data. Due to different input data, threads with odd IDs go to the if path, while threads with even IDs take the else path [Fig. 1(a)]. Fig. 1(b) shows a simple case where every thread has different loop bounds. In this case, there will be a case that the some threads execute the loop while other threads do nothing.

Control flow divergence can seriously affect the performance. However, it is not trivial to eliminate the control flow divergence. Thread regrouping based on input data ranges as described by Zhang *et al.* [7] is not accurate enough: threads with different input data may take the same execution path, and threads with identical input data may still have different execution paths due to thread IDs. Thus, we need an accurate representation of execution paths for threads. Detailed program execution paths may represent this behavior, but it may not be efficient and can be extremely long. Therefore, we need a balance: a simple technique with a compact representation that still accurately represents threads' behavior. In addition, reducing the absolute amount of control flow divergence does not necessarily reduce the performance impact of divergence: certain divergences will have higher performance impact than others. Therefore, we need an accurate estimate of the performance impact of thread regrouping together with efficient thread regrouping algorithms that can use a compact and accurate representation to guide selection of threads to group together. We will discuss our proposed performance metrics and thread regrouping algorithms in Sections IV and V, respectively.

### A. Elimination Mechanism

Zhang *et al.* [7] proposed thread-data remapping, a software-based control flow divergence elimination mechanism. The basic idea is to switch the data sets that threads work on so that threads in a warp may execute the same path. There are two options to realize this mechanism: 1) reference redirection (indirect access) and 2) data layout transformation. Fig. 2(a) and (b) illustrates the two options of thread-data remapping mechanism with an example warp size of 4.

In reference redirection, the thread's access to data memory is redirected by an indirection reference array $D[]$ that provides a new ID for all the threads. As illustrated in Fig. 2(a), the new ID is $D[\text{tid}]$, and threads will use their new IDs in accessing the data memory and continuing the computation. This mapping option can be used to eliminate both thread ID-dependent divergence and data-dependent divergence. Implementation of this mapping option in the code is by inserting additional parameter in the kernel function. Then, during kernel execution, all instances of thread ID in the kernel are replaced by the new ID. If thread ID is used as part of kernel computation, the replacement will automatically correct that portion as well. Reference redirection incurs overhead due to the addition of instructions to fetch the reference indirection. This mapping option may also hurt the memory access bandwidth. In GPUs, memory accesses from a warp are coalesced if threads in a warp perform sequential access to the memory. However, since threads get new IDs, this redirection will break the sequential access pattern, generating uncoalesced memory access as shown in Fig. 2(a).

Rather than changing the thread ID, the second mapping option is by physically reorganizing the input data as shown in Fig. 2(b). For the entire data, we rearrange the input data

[2]Formally denoted the postdominator.

based on the redirection array $D[]$, so that the $i$th item in new data is equal to $D[i]$th item of old data. Unlike reference redirection, we do not need to make any modification in the kernel function. As the data are physically rearranged, there is no change in coalescing memory access. However, implementation of this mapping option requires additional overhead to perform the input data rearrangement on the CPU before the GPU kernel call. Zhang *et al.* [7] demonstrated that data movement overhead can be hidden through pipelining kernel calls between the CPU and the GPU.

### *B. Limitations of Elimination Mechanism*

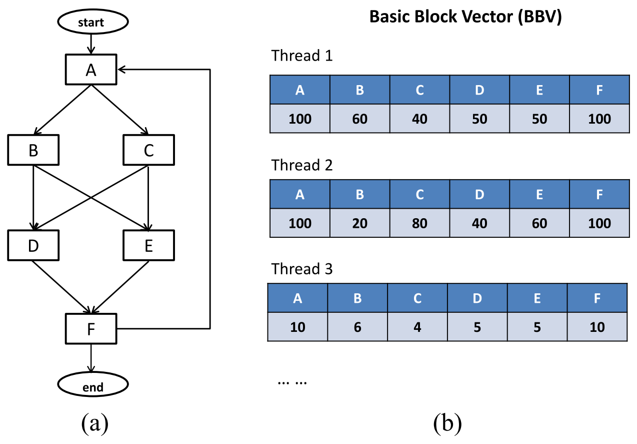
This paper focuses on control flow divergence applications, which can be eliminated via thread-data remapping. However, this mechanism is not always safe. In case where the kernel needs synchronization over the formed warps [12] or has dependencies among threads [7] that disqualify work order reorganization, thread-data remapping may destroy the synchronization and incorrectly evaluate dependencies. Such limitations exist for all thread divergence optimization techniques [7], [8], [12]. In some cases, these limitations can potentially be eliminated through programming language or compiler optimization techniques, but in other cases these limitations cannot be eliminated and must be used as a signal to not modify that portion of the code. However, the problem of determining when and how to eliminate synchronization and dependency limitations is beyond the scope of this paper.

Our techniques are limited to applications with GPU kernels that are called with sufficient frequencies and have long execution time. By eliminating the control flow divergence, the overall execution time can be significantly improved. In our system, this is not a limitation because such kernels are exactly the ones which are the most attractive targets for control flow divergence elimination.

## IV. Performance Model

Two metrics have been widely used to measure control flow divergence: 1) divergent branches and 2) divergent warps ratio. Divergent branches are measured by NVIDIA visual profiler [9]. The divergent branches count is incremented each time a branch instruction causes at least one thread in the warp to take a different execution path than the other active threads [9]. Therefore, the divergent branches count records the number of divergent branches in all warps. Instead of counting individual branch instructions, divergent warps ratio [7] measures the percentage of warps that have any control flow divergences. Both metrics quantify the amount of divergence for the GPU kernel, but (as we will demonstrate) neither metric represents the performance impact of these divergences. It is possible that both metrics indicate a significant divergence problem, but the performance is only slightly impacted (e.g., frequent single instruction divergences that do not affect performance), or vice versa.

Control flow divergence is caused by different behavior (e.g., execution path) of the threads within a warp. Thus, to estimate the performance impact of control flow divergence, we need to represent thread-specific behavior accurately. More importantly, thread-specific behavior information can be used



| A | B | C | D | E | F |
|---|---|---|---|---|---|
| 100 | 60 | 40 | 50 | 50 | 100 |

| A | B | C | D | E | F |
|---|---|---|---|---|---|
| 100 | 20 | 80 | 40 | 60 | 100 |

| A | B | C | D | E | F |
|---|---|---|---|---|---|
| 10 | 6 | 4 | 5 | 5 | 10 |

Fig. 3. (a) CFG of a program. (b) BBVs of threads.

to guide the thread regrouping to minimize the impact of control flow divergence on performance. However, divergent branches and divergent warps ratio are simple aggregate statistics; neither of them provide accurate performance estimation of control flow divergence or thread-specific information for thread regrouping.

To accurately represent thread-specific behaviors, we use BBV. A basic block is a sequential series of instructions with only one entry and one exit point. For the BBV, each thread has a set of counters, one per basic block in the kernel. Then, each time the thread touches a basic block, that counter is incremented. BBV is a compact representation of thread-specific behavior: its size depends on the static program size, not dynamic program size. This allows us to design efficient performance models and thread regrouping algorithms based on the BBV. As we will demonstrate in the experiments, our performance models based on the BBV can accurately distinguish the performance impact of control flow divergence for different thread regrouping algorithms.

Fig. 3 presents an example of control flow graph (CFG) of a GPU kernel and its BBV. The CFG consists of six basic blocks. For each thread, its BBV records the execution counts of basic blocks for it. For example, as shown in Fig. 3(b), the execution count of basic block $B$ of threads 1–3 is 60, 20, and 6, respectively. In this example, threads 1–3 take different paths at conditional branches (basic blocks $A$, $B$, $C$, and $F$), leading to different execution counts of the basic blocks for different threads.

### *A. BBV Estimation*

To obtain precise BBV for the threads, we can execute the kernel program on the target architecture. However, this will be too slow. In this paper, we apply regression modeling to efficiently obtain estimates of BBV for the threads. For each basic block $b$, we model its execution count $\mathrm{BBV}[b]$ as a weighted sum of input variables plus random noise. Let $\mathrm{BBV}[b] = \mathrm{BBV}_{t_1}[b], \ldots, \mathrm{BBV}_{t_N}[b]$, where $\mathrm{BBV}_{t_i}[b]$ denotes the execution count of basic block $b$ for thread $t_i$. $\mathrm{BBV}_{t_i}[b]$ is estimated as follows:

$$\mathrm{BBV}_{t_i}[b] = \beta_0 + \beta_1 \cdot I_{t_i}[1] + \cdots + \beta_p \cdot I_{t_i}[p] + \beta_{p+1} \cdot \mathrm{tid} \quad (1)$$

**Listing 1** Code Instrumentation

```
__global__ void kernel(int * input, ...){
{
1.    int tid = ...;
2.    ...
3.    for(int i = 0; i < input[tid]; i++) {
4.      if(tid = 1){ // instrumentation
5.           __prof_trigger(xx);
6.      }
7.    }
8.    if(input[tid] > 100){
9.        if(tid = 1){ // instrumentation
10.           __prof_trigger(xx);
11.        }
12.   }
13.
```

where $I_{t_i} = I_{t_i}[1], \ldots, I_{t_i}[p]$ denote the input variables for thread $t_i$, and tid is the thread identifer for thread $t_i$. These variables are constant for a given input workload and thread. Let $\beta = \beta_0\beta_1, \ldots, \beta_p + 1$ denote the corresponding set of regression coefficients used in describing the linear function of input variables and noise in (1). For each basic block $b$, we employ a dataflow analysis to determine the set of input variables that impact the execution count of $b$. For example, in the sample code of Listing 1, the execution count of the for loop (line 3) and if statement (line 8) of a thread depends on its data in the input array.

We fit the model in (1) using the method of least square. More clearly, for each basic block $b$, we determine the $p + 2$ coefficients (e.g., $\beta_0, \ldots, \beta_{p+1}$) based on a few observations. The least square fitting model attempts to minimize the derivation of predicted and actual value. We obtain a number of observations of BBV[$b$] through code instrumentation and profiling. For the target thread and basic block, we insert a conditional if statement and a call to the NVIDIAs profiler trigger prof_trigger [13]. Listing 1 illustrates the code instrumentation using an example. In this case, through instrumentation and profiling, we can collect the execution counts for the loop (line 3) and if statement (line 8) for thread 1. It is often that the number of inputs is smaller than the thread block size. Hence, we train the model based on the observations of the threads in one thread block and predict the BBV for the threads in other thread blocks. For example, for application MarchingCubes (MC), we train the model for all the basic blocks using the threads in one thread block (32 threads) and then predict the BBV for other threads. The mean square error is smaller than 0.001. Note that the regression model is built offline but used online. When the kernel is launched, we feed the inputs of the threads into the model as shown in (1) to derive the BBVs.

We then develop the performance metrics based on BBV. More specifically, the performance metrics leverage three components: 1) warp performance; 2) thread-block performance; and 3) kernel performance. At warp performance level, we model per-instruction latencies, execution counts of basic blocks, and control flow divergence; at thread-block performance level, we model the number of warps within a thread-block; at the kernel performance level, we model the number of SMs and thread-block scheduling.

**Listing 2** Sample of Basic Block's Instructions

```
bb_32:PC = 0xa38 mul.lo.s32 %r167, %r106, 8
bb_32:PC = 0xa40 ld.const.u32 %r168, [char_size]
bb_32:PC = 0xa48 mul.lo.u32 %r169, %r167, %r168
bb_32:PC = 0xa50 rem.u32 %r170, %r166, %r169
bb_32:PC = 0xa58 div.u32 %r171, %r170, %r168
```

## *B. Performance Metrics*

We develop two performance metrics, BBV-weighted and BBV-weighted-scheduled, which are different in terms of thread block scheduling.

*1) BBV-Weighted:* For each basic block, we can easily compute the number of instructions in it. However, each instruction has different latency; a basic block with fewer instructions may have longer latencies than a basic block with more instructions. Thus, we apply instruction-specific latencies measured using micro-benchmarking [11] and store them to a table. Once we collect the instructions of a basic block, we calculate the total instruction latency of the basic block based on the reference table. For example, Listing 2 shows the instructions of a basic block. Benchmarking results on NVIDIA GTX480 (details will be presented in Section VII) show that `mul` instruction takes 18 clocks, loading a constant variable needs 46 clock cycles, and both `rem` and `div` instructions in unsigned integer take 264 clock cycles. Thus, the latency of this basic block is 610 clock cycles.

The threads in a warp may have different behavior, leading to different execution counts of a basic block for different threads in a warp. In BBV-weighted, for each basic block, we find the maximum execution counts for any thread in the warp. For example, if one thread in warp 1 executes basic block $b$ two times, and another thread in the same warp executes basic block $b$ five times, then the maximum execution count is five. The total latency of basic block $b$ in warp $w$ is computed as the maximum execution counts of $b$ in warp $w$ multiplied by the latency of basic block $b$ as follows:

$$\mathrm{Lat}_{b|w} = \mathrm{lat}[b] \times \max_{t \in w}(\mathrm{BBV}_t[b]) \tag{2}$$

where lat[$b$] is the total latency of all the instructions in thread block $b$ and BBV$_t$[$b$] is the execution count of basic block $b$ of thread $t$. The performance of warp $w$ is then estimated as the sum of latencies of all basic blocks

$$T_w = \sum_{b \in B} \mathrm{Lat}_{b|w} \tag{3}$$

where $B$ is the set of basic blocks of the GPU kernel. Finally, the thread block latency estimate is the sum of the estimated performance of warps in the thread block (4), and the total kernel performance estimate is the sum of thread block latencies divided by the number of SMs available on the GPU (5), which corresponds to a static round-robin thread block assignment with exactly equal thread block latencies

$$T_{\mathrm{Tb}} = \sum_{w \in \mathrm{Tb}} T_w \tag{4}$$

$$T_{\mathrm{total}} = \frac{\sum T_{\mathrm{Tb}}}{\#\mathrm{SM}}. \tag{5}$$

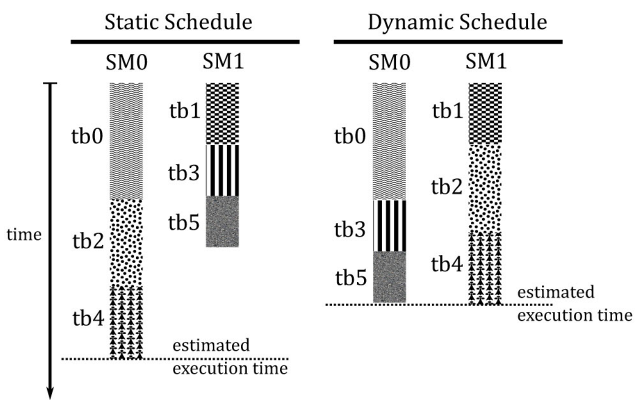


Fig. 4. Thread block schedule.

If the threads in a warp have different execution count for a basic block, then the threads within the warp must diverge. In BBV-weighted, the latency of thread block $b$ in warp $w$, $\mathrm{Lat}_{b|w}$ (2), is estimated using the maximum execution counts of $b$ in $w$. In other words, all the threads in a warp have to wait until the slowest thread completes.

*2) BBV-Weighted-Scheduled:* In BBV-weighted metric, we use the sum of thread block latencies divided by the number of SMs available as the estimate of kernel performance. This estimation assumes that there is little variance in execution time among thread blocks. However, due to control flow divergence, the execution time of warps may vary, leading to variation in thread blocks' execution time. To accurately model performance, we must therefore model the thread block scheduling algorithm employed by the hardware. The exact behavior of the scheduler has not been disclosed by NVIDIA, but previous research [14] indicated that a round-robin scheduling policy correlates well with the performance. Therefore, we develop BBV-weighted-scheduled metric, which combines the thread block performance estimation in BBV-weighted together with dynamic round-robin scheduling policy for performance estimation.

In BBV-weighted-scheduled, for each thread block, we estimate its latency using (4); the total kernel performance is estimated using dynamic round-robin policy. After the initial thread block assignment to fill all the SMs, the rest of the thread blocks will be inserted into a queue and dispatched to GPUs when there are available SMs. Fig. 4 shows a comparison of static and dynamic round-robin schedule. For example, thread block 2 is scheduled to SM0 in static schedule. However, thread block 2 is scheduled to SM1 in dynamic schedule as SM1 finishes execution earlier than SM0. The dynamic scheduling, therefore, performs a simple runtime load balancing between SMs. Note that in the example, we assume the occupancy (the number of concurrent thread blocks on one SM) is 1 for demonstration purposes. In reality, we use the occupancy calculator to determine the occupancy. If one SM can run more than one thread block in parallel, then multiple thread blocks will be assigned to it.

For the BBV-weighted and BBV-weighted-scheduled metrics, we do not intend to accurately predict the absolute performance, but to accurately estimate the performance impact of control flow divergence and the performance difference among different thread regrouping optimizations (Section V). For all the metrics, we assume that the global memory bandwidth is not the bottleneck on the performance; the long memory latency can be hidden by the provided with sufficient occupancy. As we will demonstrate, our metrics can accurately estimate performance for compute-bound control flow divergence GPU applications.

The estimation may be different from the actual performance for a few reasons. First, the BBV does not capture the exact sequence of basic block executions. Thus, the control flow divergence might be underestimated by the BBVs. Second, we assume the memory bandwidth is not bottleneck of the performance and not affected by the thread regrouping optimizations. Finally, we do not model other implementation details including instruction and warp scheduling, reconvergence points, which are not disclosed by NVIDIA. Despite these reasons, we will demonstrate that, in practice, the metrics we develop correlate well with the performance. Therefore, the metrics can be used to evaluate various control flow optimizations without actually performing the thread regrouping, and rerunning the application.

## V. Thread Regrouping Algorithms

In theory, we can enumerate all the possible thread regrouping options and compare them using our performance metrics in Section IV. However, in practice, exhaustively testing all the thread regrouping options is infeasible as the number of options grows exponentially with the number of threads of the kernel. Hence, in the following, we propose low-complexity thread regrouping algorithms that can be used online to efficiently derive an optimized thread regrouping solution.

At a high level, we wish to group together threads that behave similarly—optimally with no control flow divergence, but with minimized performance impact if control flow divergence is unavoidable. To achieve this goal, we use per-thread BBVs to represent thread behavior. We will examine three alternative thread regrouping algorithms, Sorting, Greedy, and Greedy Max. Moreover, these three algorithms will be used to evaluate the accuracies of our performance metrics.

Our thread regrouping algorithms first divide $N$ threads into $K$ groups. The size of the group is then calculated by groupsize $= \lceil N/K \rceil$. In situations where $\lceil N/K \rceil$ is not a multiple of warp size, there will be warps that are formed by two different groups and potentially will suffer from control flow divergence. In order to avoid such situation, we set a fixed group size to be multiple of warp size: 32, 64, and so on. Thus, the number of groups, $K$, is calculated by $\lceil N/\text{groupsize} \rceil$. In the following, we present the details of our thread regrouping algorithms.

### *A. Sorting Algorithm*

Our first thread regrouping algorithm is Sorting. We compare the threads based on the per-thread BBV. Each BBV is a vector of integers. We sort the BBVs in lexicographical order. More clearly, we compare element by element in two BBVs that have the same index until one element is not equivalent to the other. The comparison result of two BBVs in lexicographical order is the comparison result of the first nonmatching elements. For example, $\mathrm{BBV}_{t1} = \{1, 20, 10, 20, 1\}$ is smaller than $\mathrm{BBV}_{t2} = \{1, 30, 20, 30, 1\}$. After sorting, every groupsize consecutive threads form a group, where the last group may be smaller if $N$ threads cannot be equally divided into $K$ groups.

### *B. Greedy Algorithm*

Our Greedy algorithm is an iterative method. At the beginning, each thread forms an individual group. In each iteration, we calculate the estimated performance gain (described below) through merging and merge the two groups with the maximal gain. If the merged group size exceeds the groupsize, then the first groupsize threads form one group and the remaining threads in the groups form a new group and will be used for further merging. This process is performed iteratively until the number of groups is equal to $K$.

To compute the estimated performance gain, we first define the minimal and maximal execution counts of each basic block $b$ within a thread group $g$ as follows:

$$\mathrm{Min}_g^b = \min_{t\in g} \mathrm{BBV}_t[b] \tag{6}$$

$$\mathrm{Max}_g^b = \max_{t\in g} \mathrm{BBV}_t[b]. \tag{7}$$

Intuitively, we want to merge two thread groups which behave similarly to minimize any potential negative impact due to control flow divergence. Thus, we merge two thread groups whose BBVs have similar (preferably identical) execution counts for each basic block. As all the threads that follow the same BBV (or portion thereof) can simultaneously make progress, merging these threads into a group increases computation throughput with no change to total latency. Thus, we define the benefit of merging two thread groups $g_1$ and $g_2$ as follows:

$$\mathrm{Benefit}(g_1, g_2) = \sum_{b\in B}\Big(\mathrm{lat}[b] \times \mathrm{Min}_{g_1\cup g_2}^b\Big). \tag{8}$$

We do not want to merge two thread groups if there are significant differences in their BBVs since merging these two thread groups will cause the total warp latency to increase due to control flow divergence. Thus, we define the cost of merging two threads groups $g_1$ and $g_2$ as follows:

$$\mathrm{Cost}(g_1, g_2) = \sum_{b\in B}\Big(\mathrm{lat}[b] \times \Big(\mathrm{Max}_{g_1\cup g_2}^b - \mathrm{Min}_{g_1\cup g_2}^b\Big)\Big). \tag{9}$$

We estimate the number of executions where two groups can execute together as benefit, but the number of executions where they diverge as cost. For example, if one group executes basic block $b$ once and the other group executes basic block $b$ twice, we estimate that the first time they can execute together as benefit, but the second execution as cost. This is because during the second execution, one group is diverged, waiting for the other group and making no forward progress. Finally, we define the overall gain as follows:

$$\mathrm{Gain}(g_1, g_2) = \mathrm{Benefit}(g_1, g_2) - \mathrm{Cost}(g_1, g_2). \tag{10}$$

For the threads with identical BBVs, merge provides benefit with no cost, thus guaranteeing that identical threads will be merged together before adding nonidentical threads. The details of our Greedy solution are presented in Algorithm 1.

**Algorithm 1** Pseudocode for the Greedy Regrouping

```
1: Let $G$ be the set of thread groups
2: Let $E$ be the set of edges among groups
3: for every $g_i, g_j \in G$ do
4:     calculate gain($g_i, g_j$)
5: end for
6: groups = 0;
7: while ( groups < $K$ - 1) do
8:     find edge($g_m, g_n$) with the maximal gain
9:     merge($g_m, g_n$) as $g_{new}$
10:    for every group $g \in G$ do
11:        remove edge ($g_m, g$) from $E$
12:        remove edge ($g_n, g$) from $E$
13:        insert edge ($g_{new}, g$) into $E$
14:        calculate gain($g_{new}, g$)
15:    end for
16:    remove $g_m, g_n$ from $G$
17:    insert $g_{new}$ to $G$
18: end while
Function: Merge
19: function MERGE($g_m, g_n$)
20:    if size($g_m$) + size($g_n$) >= groupsize then
21:        $g_{new} = g_m \cup g_n$
22:        the first groupsize threads form the group, $g^*$.
23:        $g_{new} = g_{new} - g^*$
24:        groups = groups + 1
25:    else
26:        $g_{new} = g_m \cup g_n$
27:    end if
28: end function
```

### *C. Greedy-Max Algorithm*

Greedy algorithm tends to form a group of threads (e.g., 32 threads) using similar threads. However, it does not optimize the thread block ordering. The ordering of thread blocks has an impact on load balancing among the SMs and thus affects the total kernel execution latency under the dynamic round-robin thread block scheduling policy.

Fig. 5 illustrates how the ordering of thread blocks affects the total kernel execution latency. In Fig. 5(a), there are two different thread block queues: 1) queue Q1 is an unordered queue and 2) queue Q2 is a latency-based order queue (thread blocks are sorted in descending order of their latency). Fig. 5(b) and (c) illustrates the round-robin scheduling policy that assigns the thread blocks to the available SMs for Q1 and Q2, respectively. We assume the maximum number of thread blocks that can simultaneously execute on an SM is 1. In Fig. 5(b), the assignment is scheduled as follows: at time 0 ($t0$), thread block 0 (tb0) is assigned to SM0, tb1 is assigned to SM1, and tb2 is assigned to SM2. Since each SM can only handle one thread block at a time, the rest of thread blocks will be queued and wait for the first available SM. Then, tb1 in SM1 is done at $t1$, and thus instantly tb3 is assigned to SM1. When tb2 is complete at $t2$, tb4 is assigned to SM2. This scheduling mechanism is repeated until no more thread blocks left in the queue. However, if we apply the same scheduling mechanism to Q2 [Fig. 5(c)], we find that the workload is more balanced among SMs compared to Q1 and the final execution time of Q2 is less than Q1. Therefore, the ordering of thread blocks in the kernel has an impact on the overall performance.

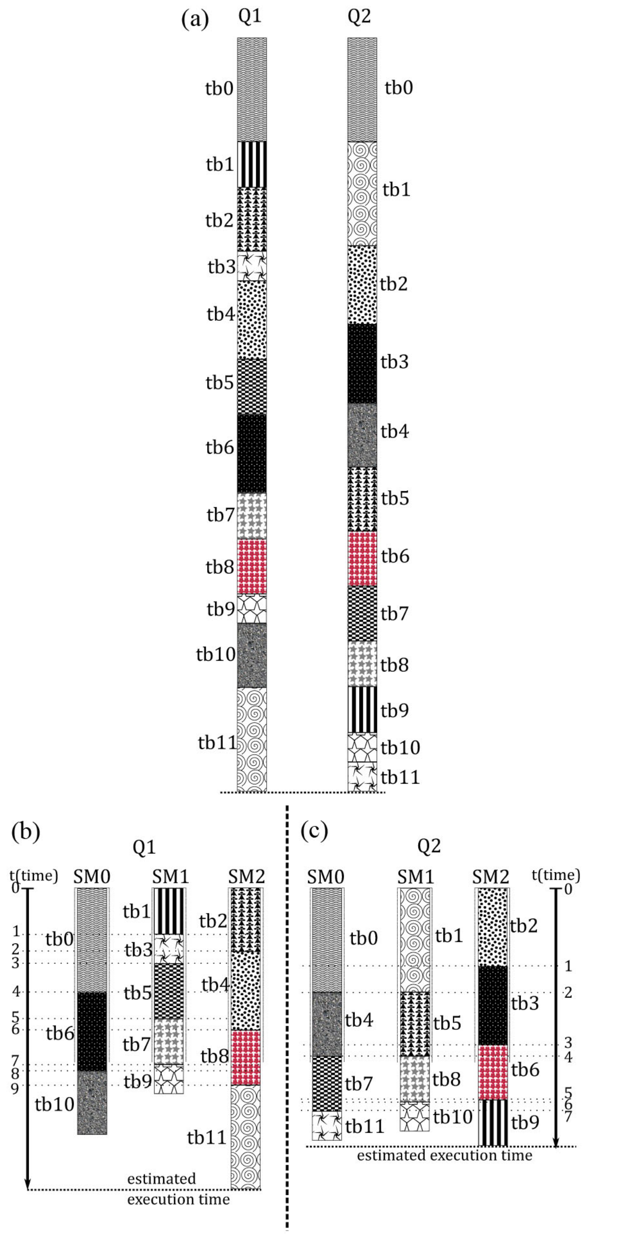


Fig. 5. Impact of thread block order to load balancing in SMs. (a) Ordered and unordered thread block queue. (b) Round-robin schedule with unordered queue. (c) Round-robin schedule with ordered queue.

In our Greedy-Max algorithm, we still form groups of threads using overall gain metric (10), but we form the groups in the descending order of their latency. We define the latency of thread $t$ as follows:

$$\text{Lat}_t = \sum_{b \in B} \text{lat}[b] \times \text{BBV}_t[b] \quad (11)$$

where $B$ is the set of basic blocks of the GPU kernel.

The details of Greedy-Max are described in Algorithm 2. We start with an empty thread group $g$. Initially, $g$ is filled with the thread that has the maximum latency ($\text{Lat}_{t'} = \max_{t \in T} \text{Lat}_t$). Then, $g$ is filled by either a thread that has identical BBV to an existing thread in $g$, or a thread that gives maximum performance gain if merged with $g$. This step is repeated until the size of $g$ is equal to groupsize. By doing this iteratively, we can build $K$ groups. Compared to Greedy algorithm,

**Algorithm 2** Pseudocode for Greedy-Max

```
1: Let $T$ be the per-thread $BBV$s list
2: Let $G$ be the set of thread groups and initially empty
3: Let $g$ is an empty thread group
4: while ( size($G$) < $K$) do
5:     let $t'$ s.t. $Lat_{t'} = max_{t \in T} Lat_t$
6:     remove $t'$ from $T$ and add it to $g$
7:     while ( size($g$) < $groupsize$) do
8:         initial $max\_gain$
9:         $identical = False$
10:        $i = 0$
11:        while (not $identical$ and $i <$ size of $T$) do
12:            if $t_i \in T$ match with any $j \in g$ then
13:                $identical = True$
14:                $max\_t = t_i$
15:            else
16:                calculate $cur\_gain$ of edge($g, t_i$)
17:                if $cur\_gain > max\_gain$ then
18:                    $max\_gain = cur\_gain$
19:                    $max\_t = t_i$
20:                end if
21:            end if
22:            $i = i + 1$
23:        end while
24:        add $max\_t$ into $g$
25:        remove $max\_t$ from $T$
26:    end while
27:    add $g$ into $G$
28:    clear $g$
29: end while
```

Greedy-Max achieves load balancing via thread block ordering as shown in Fig. 5.

## D. Discussion

The Sorting algorithm regroups the threads by sorting the per-thread BBVs in lexicographical order. By doing this, it tends to group similar threads. However, it does not consider the differences between basic blocks and model the warp performance and thread block scheduling. Based on the warp performance estimation by (2) and (3) in Section IV, Greedy algorithm models the differences between basic blocks, and groups threads together using the gain metric (10). However, it ignores the thread block latency and their ordering, which affects the load balancing among the SMs and the total latency. Our final thread group algorithm, Greedy-Max, not only models the thread performance and considers the thread similarities using gain metric, but also prioritizes the threads with high latency. Based on the dynamic round-robin thread block scheduling policy, ordering thread blocks in descending order of their latencies is helpful for load balancing among the SMs and optimizes the total latency. In the experiments, we will compare the performance speedup of the three thread regrouping algorithms.

## VI. Divergence Optimization Framework

Our control flow divergence optimizations relies on synergistic collaboration of compile-time and runtime components as shown in Fig. 6. At compile-time, we analyze the kernels, build the BBV estimation model, and enable the control flow divergence using the elimination mechanism described in Section III. At runtime, we predict per-thread BBVs and call thread regrouping algorithms to regroup the threads.

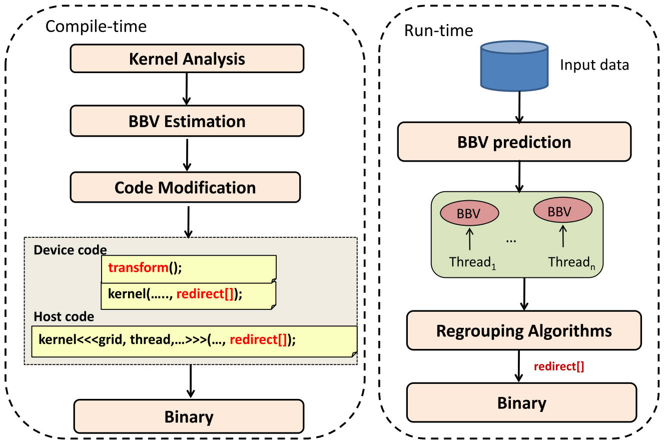


Fig. 6. Divergence optimization overview.

### A. Compile Time

Given a GPU kernel program, we first parse the kernel code, build the CFG at the source code level, and profile the latency for every instruction. Then, we build the regression model for BBV estimation for each basic block as described in Section IV-A. Each GPU program contains a few inputs available offline. We profile the kernel using the profiler trigger [13] as shown in Listing 1. In general, one GPU kernel will launch thousands or even more number of threads. But, we observe that it is sufficient to profile the threads only in one thread block to build the BBV estimation model. Note that the BBV estimation model is built at compile-time but used at runtime. At runtime, when the kernel is called with new inputs, we predict the BBVs using the BBV estimation model built offline.

We enable the divergence optimization through code transformation. Both the host and device code have to be slightly modified. For the host code, we insert the thread regrouping function before the kernel call. For example, as shown in Fig. 6, if the reference redirection mechanism is used, the redirect[] array is inserted and threads read their input data based on the redirect[] array [Fig. 2(a)]; if the data layout transformation mechanism is used, the transform function is called to reorganize the data. But, the redirect[] array or data layout transformation order are generated online by the thread regrouping algorithms. Finally, we compile the modified source code to binary.

### B. Runtime

When the kernel is launched with its input data, we predict the per-thread BBV using the BBV estimation model. Then, we use the per-thread BBV to represent the thread behavior and invoke the thread regrouping algorithms to produce the new thread grouping for improved control flow divergence and application performance. Meanwhile, based on the per-thread BBV, we can estimate the performance improvement potential using the performance models developed in Section IV. If the thread regrouping leads to negative or very marginal performance improvement, then the thread regrouping will be disabled.

The overhead incurred at runtime includes two parts: 1) the overhead of the per-thread BBVs prediction using the BBV estimation model and 2) the overhead of thread regrouping algorithms. These overheads are very small compared to the GPU kernel time.

## VII. Experiments

### A. Experiments Setup

Table I shows the list of benchmark applications used. We collect benchmarks from different sources that exhibit control flow divergence. We include two real-world applications—3-D sound localization (SL) [3] and stereo matching (SM) [20], which both exhibit complex control flow divergence. For each application, Table I gives the transformation elimination mechanism (reference redirection or data layout transformation), the total number of threads, and performance bottleneck. Performance bottleneck (memory-bound or compute-bound) is identified by using the NVIDIA visual profiler [9]. In our optimization framework, we rely on profiling to train the BBV estimation model. We evaluate our techniques using different profiling and evaluation inputs.

MC benchmark is taken from the CUDA SDK. MC implements marching cubes algorithm, which computes geometric iso-surface from a 3-D volume data set. Each parallel thread calculates one voxel (cube). Divergence in MC is caused by different threads calculating different number of vertices for each cube vertex index. Smith-Waterman (SW), CUDA-EC (EC), Needleman-Wunsch (NW), and MUMmer (MUM) benchmarks are all bioinformatics applications. SW aligns a query polypeptide sequence against a protein database, while MUM aligns a set of DNA queries against a single reference sequence. EC benchmark corrects errors in short reads from high-throughput sequencing. NW benchmark calculates global alignment scores between a DNA query and reads from DNA database. Divergences of those four benchmarks are caused by diversity in DNA or protein sequence content being processed by threads. 3-D SL finds the direction of the sound captured by the acoustic microphone in 3-D fashion [3]. SL exhibits control flow divergence because different threads are assigned with different workload (e.g., different number of 3-D angles). SM is an important underlying technology for 3-D-video application [20]. SM contains thread divergence because the grid building kernel selects global maxima based on the cost value which differs among threads.

Although it seems intuitive that there are many benchmarks that exhibit control flow divergence, few freely available benchmarks currently demonstrate this sort of behavior. We examined the benchmarks including the set of applications used for prior control flow divergence studies and all the applications that demonstrate control flow divergence from the CUDA SDK.

Control flow divergence elimination is important for all the generations of GPUs. In this paper, we use the widely used NVIDIA Fermi-based GPUs to evaluate our techniques. In particular, all the experiments are performed on NVIDIA GTX480 GPU. The specification of our evaluation machine is shown in Table II. Using micro-benchmarks [11], we collect the instruction latencies on the GTX480. Table III presents

TABLE I
BENCHMARK CHARACTERISTIC

| **Benchmarks** | **Source** | **Description** | **Input** | **Elimination Mechanism** | **Threads** | **Performance Bottleneck** |
|---|---|---|---|---|---|---|
| Marching Cubes (MC) | CUDA SDK | calculates geometric iso-surface from a volume dataset [15] | float and int arrays | reference redirection and data layout transformation | 262144 | Compute-bound |
| Smith-Waterman (SW) | [16] | local sequence alignment algorithm | 8k protein sequences | data layout transformation | 32768 | Compute-bound |
| CUDA-EC (EC) | [17] | sequence error correction | 6k sequences | data layout transformation | 24576 | Memory-bound |
| Needleman-Wunsch (NW) | [18] | global sequence alignment algorithm | 2k sequences | data layout transformation | 32768 | Compute-bound |
| MUMmer (MUM) | [19] | fast genome alignment tool | 16k sequences | data layout transformation | 65536 | Memory-bound |
| 3D Sound Localization (SL) | [3] | finds the direction of captured sound in 3D | 36m float array | data layout transformation | 17920 | Compute-bound |
| Stereo Matcher (SM) | [20] | depth map based stereo matcher | two images 450x375 px | reference redirection | 194560 | Compute-bound |

TABLE II
SPECIFICATION OF TESTING MACHINE

| | |
|---|---|
| **GPU** | NVIDIA GTX 480 |
| **Architecture** | Fermi |
| **SMs (Cores per SM)** | 15 (32) |
| **Total CUDA Cores** | 480 |
| **GPU Clock per core** | 1.4 GHz |
| **GPU RAM** | 1.5 GB |
| **GPU Memory Clock** | 1.85 GHz |
| **CPU** | Intel i5-750 @ 2.67 GHz |
| **CPU RAM** | 4 GB |

TABLE III
INSTRUCTION LATENCY (IN CLOCKS)

| Operation | **Type** | | | |
|---|---|---|---|---|
| | **uint** | **int** | **float** | **double** |
| add, sub, mul | 18 | 18 | 18 | 24 |
| div | 264 | 300 | 984 | 1188 |
| min, max | 36 | 20 | 36 | 48 |
| mad | 20 | 20 | 20 | 24 |
| and, or, shl, shr | 18 | 18 | - | - |
| xor | 1 | 18 | - | - |
| rem | 264 | 297 | - | - |
| sinf, cosf | - | - | 40 | - |
| tanf | - | - | 116 | - |
| abs | - | 36 | 36 | - |
| tex | 220 | 220 | 220 | - |
| sqrt | - | - | 208 | - |
| rsqrt | - | - | 70 | - |
| rcp | - | - | 228 | - |
| __expf() | - | - | 106 | - |
| __exp2f() | - | - | 88 | - |
| __logf(), __log10f() | - | - | 88 | - |
| __log2f() | - | - | 70 | - |
| __powf() | - | - | 110 | - |
| __umul24() | 36 | - | - | - |
| __mul24() | - | 36 | - | - |
| __fadd_rn(),__fadd_rz(), __fmul_rn(),__fmul_rz() | - | - | 18 | - |
| __fdividef() | - | - | 88 | - |
| __dadd_rn() | - | - | - | 24 |
| __syncthreads | 16 | - | - | - |
| Operation | **Memory Type** | | | |
| | **local** | **constant** | **global** | **shared** |
| load | - | 46 | - | 44 |
| store | - | - | - | 44 |

the latency results. The latency table will be used for computing the total latency of basic blocks for each application. GTX480 contains caches. The latencies to constant and shared memory are constant as they are always hits. However, for loads/stores to local and global memory, the latencies depend on their cache behavior (e.g., hits or misses). We empirically measure the cache hit ratio using NVIDIA profiler and then compute an average memory access latency using the cache hit ratio.

Each benchmark is compiled using NVIDIA's CUDA compiler, nvcc. Each basic block contains a list of instructions. We compute the total latency of each basic block by summing the latencies of the instructions in it. With BBVs and each basic block's latency as input, we can estimate the performance impact of control flow divergence. Then, each application is transformed using either the reference redirection transformation or the data layout transformation. Finally, we execute both initial (without transformation) implementation and transformed implementation. We record the execution time for both implementations and gather all performance statistics including divergent branches, divergent warps, and estimated performance computed by our metrics. For the transformed implementation, we include the runtime overhead in its measurement.

In the following, we perform two sets of experiments: first, we compare the prior metrics to our proposed metrics in terms of correlation to actual performance (Section VII-B); second, we present the performance speedup of our optimization framework (Section VII-C).

## B. Metric Accuracy

As discussed in Section IV, divergent warps measure the percentage of warps that have any control flow divergence

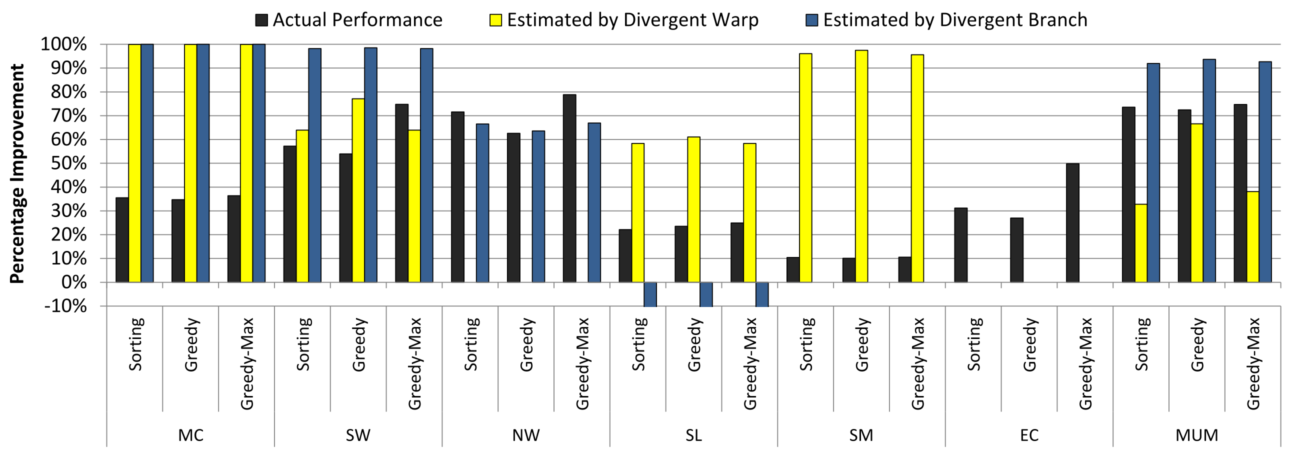


Fig. 7. Comparison of divergent warps and divergent branches metrics and actual performance improvement.

TABLE IV
DIVERGENT WARPS AND DIVERGENT BRANCHES

| **Benchmarks** | **Divergent warps / Total warps** | **Divergent branches / Total branches** |
|---|---|---|
| MC | 8192 / 8192 | 47380 / 151820 |
| SW | 1024 / 1024 | 29783 / 83517722 |
| EC | 768 / 768 | 603954 / 52079573 |
| NW | 1024 / 1024 | 97049370 / 1094483922 |
| MUM | 2048 / 2048 | 371967 / 1640166 |
| SL | 36 / 560 | 102 / 163624 |
| SM | 4760 / 6080 | 258875 / 3570480 |

TABLE V
ESTIMATION ERROR (IN %)

| Benchmark | Algorithms | BBV-weighted | BBV-weighted-scheduled |
|---|---|---|---|
| MC | Sorting | 20.9 | 7.6 |
| | Greedy | 20.1 | 6.2 |
| | Gr.-Max | 21.8 | 7.5 |
| SW | Sorting | 20.2 | 15.8 |
| | Greedy | 21.9 | 13.0 |
| | Gr.-Max | 2.6 | 7.5 |
| NW | Sorting | 5.8 | 1.4 |
| | Greedy | 7.5 | 2.0 |
| | Gr.-Max | 1.5 | 3.5 |
| SL | Sorting | 19.6 | 4.0 |
| | Greedy | 22.2 | 11.3 |
| | Gr.-Max | 22.0 | 7.2 |
| SM | Sorting | 1.5 | 0.5 |
| | Greedy | 1.4 | 0.9 |
| | Gr.-Max | 1.6 | 4.3 |
| **Average Error** | | **12.7** | **6.2** |

while divergent branches count individual branch instructions that cause divergences. Note that both prior metrics (i.e., divergent warps and divergent branches) and our metrics (i.e., BBV-weighted and BBV-weighted-scheduled) are not intended to predict the absolute performance, but to estimate the performance difference between different thread regrouping algorithms. For each application, we use each of our three regrouping algorithms, with the $K$ (the number of groups) set to 32 (a warp size), and compare the actual performance improvement on GTX480 in percentage to the performance improvement in percentage predicted by the metrics.

We first examine the prior metrics. Table IV lists the divergent warps, divergent branches, total warps, and total branches for all the benchmarks. As shown, for some applications, all the warps are divergent. For each application, we first measure the actual performance, divergent warps, and divergent branches of the original implementation (without thread regrouping). Then, for each application and thread regrouping algorithm pair, we measure the actual performance, divergent warps, and divergent branches after applying the thread regrouping optimization. Fig. 7 shows the percentage improvement of actual performance, divergent warps, and divergent branches over the initial implementation without transformation for each combination of the applications and thread regrouping algorithms. We observe that divergent warps and divergent branches give inaccurate estimation for most applications. For example, for the NW benchmark, divergent warps estimate no improvement for all the algorithms, but actual performance improvement is about 70%; for the SL benchmark, divergent branches give negative improvement for all the algorithms, but actual performance improvement is about 20%. Our results in Fig. 7 demonstrate that divergent warps and divergent branches cannot accurately estimate the performance impact of control flow divergence.

Now we examine our proposed metrics that estimate the performance impact of control flow divergence: BBV-weighted and BBV-weighted-scheduled. We separate the discussion for compute-bound and memory-bound (Section VII-D) applications. For each application and thread regrouping algorithm pair, we estimate the performance using our proposed metrics. We compare the estimation of the initial implementation and the implementation with divergence optimization. Fig. 8 presents the same performance data as in Fig. 7, and compares our proposed metrics for five

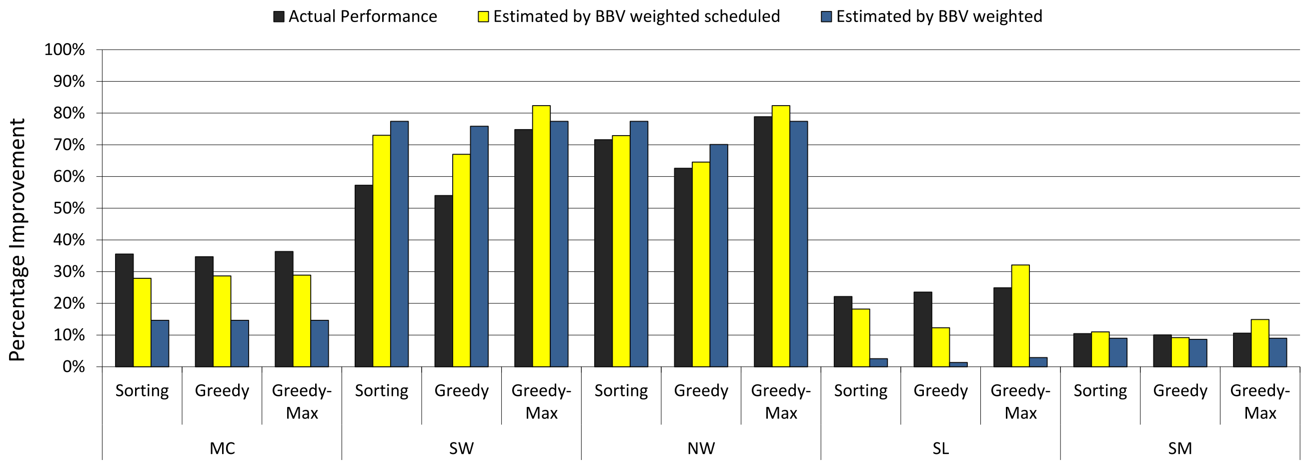


Fig. 8. Comparison of our proposed metrics and actual performance improvement for compute-bound applications.

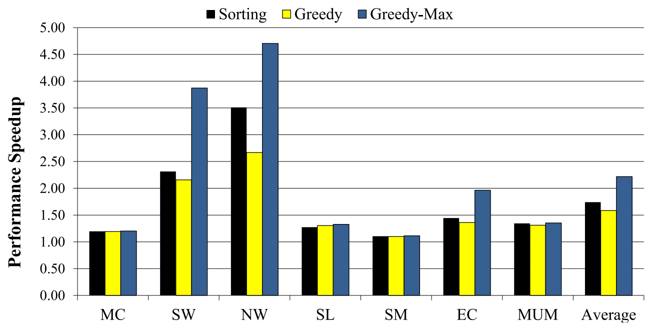


Fig. 9. Speedup of divergence optimization.

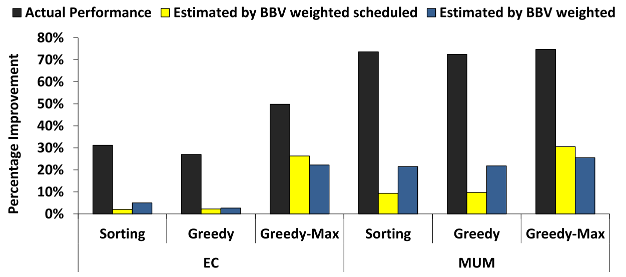


Fig. 10. Comparison of our proposed metrics and actual performance improvement for memory-bound applications.

compute-bound applications. For each metric, we show the improvement in percentage. As shown in Fig. 8, our metrics BBV-weighted and BBV-weighted-scheduled predict the performance well. Specifically, BBV-weighted-scheduled is more accurate than BBV-weighted as it considers the latency variation among thread blocks and models the thread block scheduling. Table V presents the estimation error of each metric for all the benchmarks. On average, BBV-weighted has 12.7% error, while BBV-weighted-scheduled improves the estimation error to 6.2%. Our BBV-weighted-scheduled metric is correlated well with the actual performance. This suggests that programmers can rely on these metrics to estimate the performance impact of control flow divergence and the effectiveness of the thread grouping algorithms.

## C. Performance of Divergence Optimization

Fig. 9 shows the performance speedup of our divergence optimization. We compare the performance of initial (without transformation) implementation and optimized implementation enabled by code transformation. For each application, we compare three different regrouping algorithms (e.g., Sorting, Greedy, and Greedy-Max). As indicated by our performance metrics in Fig. 8, applications MC, SL, and SM have small performance improvement. Among the three thread regrouping algorithms, Greedy-Max is the most effective algorithm. In particular, the Sorting algorithm achieves up to 3.5× speedup (on average 1.7×); the Greedy algorithm achieves up to 2.7× speedup (on average 1.6×); and the Greedy-Max algorithm achieves up to 4.7× speedup (on average 2.2×). Greedy-Max achieves the most speedup due to its accurate performance modeling of control flow divergence impact on warps and thread block scheduling.

## D. Memory-Bound Applications

When computation is not the critical path, computation optimization through thread regrouping algorithms may not lead to expected overall performance improvement. Fig. 10 presents the percentage improvement of actual performance and our metrics for the two memory-bound applications EC and MUM. For these two applications, the performance is limited by the achieved memory bandwidth. However, our metrics do not model the factors that affect memory bandwidth including memory coalescing, bank conflicts, and cache conflicts and thus are not accurate.

## VIII. Related Work

In the recent years, we have witnessed the success of GPUs for high performance computing. However, performance tuning for GPUs was not a trivial task [21]. Analytical performance models have been proposed to predict the performance improvement or identify the performance bottlenecks [4], [10], [22]–[24]. Optimization techniques have also been proposed to tailor the applications to the underlying architectural features and improve the resource utilization. The state-of-the-art GPU performance optimization techniques focused on thread and warp scheduling, cache optimization, register allocation optimization, power and aging optimization, multitasking, etc. [25]–[32].

If the threads in a warp take different path at conditional branches, then execution of different paths have to be serialized in the SIMD architecture. This is well known as control flow divergence, which has significant impact on the application performance. There are both software and hardware solutions for control flow divergence optimization.

### A. Hardware Divergence Management

Fung *et al.* [5] proposed a hardware implementation to dynamically regroup the threads to improve the SIMD utilization. Their technique allows diverged threads with the same PC but originally from different warps run together as a wider SIMD group. Meng *et al.* [6] introduced dynamic warp subdivision (DWS) as an integrated framework for both control and memory divergence. DWS selectively divides warps into warp-splits and creates separate schedulers so that each warp-split can execute independently. By doing this, DWS enables divergent threads execute together and hide each other's latency. Rhu and Erez [33] proposed the dual-path execution model that extends the reconvergence stack model to support interleaved execution of different paths. There are also proposals that optimize the SIMD divergence through compaction [34], [35]. However, the above hardware-based techniques require hardware changes and inevitably add complexity and hardware cost in the register file, scheduling logic, etc.

### B. Software Divergence Management

Besides hardware-based techniques, software-based techniques for optimizing control flow divergence have also been proposed. Software-based techniques can be applied to real GPUs without hardware changes. Zhang *et al.* [7], [8] proposed to eliminate control-flow divergence through reference redirection or data layout transformation. However, their techniques simply assume that all instances of divergence have the same impact on the performance and use the percentage of divergent warps [7] or percentage of divergent branches [9] as their metrics for estimating performance improvement. However, these metrics are not accurate for estimating performance impact of divergence as different divergences have different numbers of instructions, latencies, and also thread block scheduling can impact the overlapping of threads' execution. In contrast, we build a more accurate model by modeling these architectural features. Finally, Lee *et al.* [36] implemented compiler algorithms to handle divergence using predicates at the intermediate representation level. In contrast, our optimization can be enabled through small code changes at source code level.

## IX. Conclusion

Control flow divergence can significantly affect the performance of GPU applications. In this paper, we first present control flow divergence metrics that can accurately estimate the performance impact of control flow divergence for compute-bound GPU kernels. Our proposed BBV-weighted-scheduled metric achieves high accuracy (6.2% error) for these workloads. Thus, developers can rely on our metric to evaluate potential control flow divergence optimizations and predict whether a transformed program would have improved performance without actually performing the transformation of the application. Then, we represent threads' behavior using per-thread BBVs and develop three thread regrouping algorithms for control flow divergence optimization. We demonstrate that our divergence optimization can reduce the impact of control flow divergence, with good correlation between the actual performance impact and the impact predicted by our metrics. Our best thread regrouping algorithm Greedy-Max achieves up to 4.7× speedup (on average 2.2×).

## References

[1] V. Bertacco *et al.*, "On the use of GP-GPUs for accelerating compute-intensive EDA applications," in *Proc. Conf. Design Autom. Test Europe (DATE)*, Grenoble, France, 2013, pp. 1357–1366.

[2] Y. Deng, B. D. Wang, and S. Mu, "Taming irregular EDA applications on GPUs," in *IEEE/ACM Int. Conf. Comput.-Aided Design Tech. Dig. Papers (ICCAD)*, San Jose, CA, USA, 2009, pp. 539–546.

[3] Y. Liang *et al.*, "Real-time implementation and performance optimization of 3D sound localization on GPUs," in *Proc. Conf. Design Autom. Test Europe (DATE)*, Dresden, Germany, 2012, pp. 832–835.

[4] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W.-m. W. Hwu, "An adaptive performance modeling tool for GPU architectures," in *Proc. 15th ACM SIGPLAN Symp. Princ. Pract. Parallel Program. (PPoPP)*, Bengaluru, India, 2010, pp. 105–114.

[5] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic warp formation and scheduling for efficient GPU control flow," in *Proc. 40th Annu. IEEE/ACM Int. Symp. Microarch. (MICRO)*, Chicago, IL, USA, 2007, pp. 407–420.

[6] J. Meng, D. Tarjan, and K. Skadron, "Dynamic warp subdivision for integrated branch and memory divergence tolerance," in *Proc. 37th Annu. Int. Symp. Comput. Archit. (ISCA)*, Saint-Malo, France, 2010, pp. 235–246.

[7] E. Z. Zhang, Y. Jiang, Z. Guo, and X. Shen, "Streamlining GPU applications on the fly: Thread divergence elimination through runtime thread-data remapping," in *Proc. 24th ACM Int. Conf. Supercomput. (ICS)*, Tsukuba, Japan, 2010, pp. 115–126.

[8] E. Z. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. Shen, "On-the-fly elimination of dynamic irregularities for GPU computing," in *Proc. 16th Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS XVI)*, Newport Beach, CA, USA, 2011, pp. 369–380.

[9] *NVIDIA Visual Profiler*. [Online]. Available: https://developer.nvidia.com/nvidia-visual-profiler

[10] S. Hong and H. Kim, "An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness," in *Proc. 36th Annu. Int. Symp. Comput. Archit. (ISCA)*, Austin, TX, USA, 2009, pp. 152–163.

[11] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying GPU microarchitecture through microbenchmarking," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, White Plains, NY, USA, 2010, pp. 235–246.

[12] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic warp formation: Efficient MIMD control flow on SIMD graphics hardware," *ACM Trans. Archit. Code Optim.*, vol. 6, no. 2, 2009. Art. ID 7.

[13] *NVIDIA CUDA Programming Guide*. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/

[14] S. Ryoo, "Program optimization strategies for data-parallel many-core processors," Ph.D. dissertation, Dept. Electr. Comput. Eng., Univ. Illinois Urbana–Champaign, Champaign, IL, USA, 2008.

[15] W. E. Lorensen and H. E. Cline, "Marching cubes: A high resolution 3D surface construction algorithm," *SIGGRAPH Comput. Graph.*, vol. 21, no. 4, pp. 163–169, 1987.

[16] G. M. Striemer and A. Akoglu, "Sequence alignment with GPU: Performance and design challenges," in *Proc. IEEE Int. Symp. Parallel Distrib. Process. (IPDPS)*, Rome, Italy, 2009, pp. 1–10.

[17] *CUDA-EC.NVIDIA Tesla Bio Bench*. [Online]. Available: http://www.nvidia.com/object/ec_on_tesla.html

[18] T. R. P. Siriwardena and D. N. Ranasinghe, "Accelerating global sequence alignment using CUDA compatible multi-core GPU," in *Proc. 5th Int. Conf. Inf. Autom. Sustain. (ICIAFs)*, Colombo, Sri Lanka, 2010, pp. 201–206.

[19] M. C. Schatz, C. Trapnell, A. L. Delcher, and A. Varshney, "High-throughput sequence alignment using graphics processing units," *BMC Bioinformat.*, vol. 8, p. 474, Dec. 2007.

[20] D. Min, J. Lu, and M. N. Do, "A revisit to cost aggregation in stereo matching: How far can we reduce its computational redundancy?" in *Proc. IEEE Int. Conf. Comput. Vis. (ICCV)*, Barcelona, Spain, 2011, pp. 1567–1574.

[21] Y. Yang, P. Xiang, J. Kong, and H. Zhou, "A GPGPU compiler for memory optimization and parallelism management," in *Proc. 31st ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, Toronto, ON, Canada, 2010, pp. 86–97.

[22] L. Ma, K. Agrawal, and R. D. Chamberlain, "Theoretical analysis of classic algorithms on highly-threaded many-core GPUs," in *Proc. 19th ACM SIGPLAN Symp. Princ. Pract. Parallel Program. (PPoPP)*, Orlando, FL, USA, 2014, pp. 391–392.

[23] L. Ma, K. Agrawal, and R. D. Chamberlain, "Analysis of classic algorithms on GPUs," in *Proc. 12th ACM/IEEE Int. Conf. High Perform. Comput. Sim. (HPCS)*, Bologna, Italy, 2014, pp. 65–73.

[24] Z. Cui, Y. Liang, K. Rupnow, and D. Chen, "An accurate GPU performance model for effective control flow divergence optimization," in *Proc. IEEE 26th Int. Parallel Distrib. Process. Symp. (IPDPS)*, Shanghai, China, 2012, pp. 83–94.

[25] V. Narasiman *et al.*, "Improving GPU performance via large warps and two-level warp scheduling," in *Proc. 44th Annu. IEEE/ACM Int. Symp. Microarch. (MICRO)*, Porto Alegre, Brazil, 2011, pp. 308–317.

[26] M. Gebhart *et al.*, "Energy-efficient mechanisms for managing thread context in throughput processors," in *Proc. 38th Annu. Int. Symp. Comput. Archit. (ISCA)*, Portland, OR, USA, 2011, pp. 235–246.

[27] X. Xie, Y. Liang, G. Sun, and D. Chen, "An efficient compiler framework for cache bypassing on GPUs," in *Proc. Int. Conf. Comput.-Aided Design (ICCAD)*, San Jose, CA, USA, 2013, pp. 516–523.

[28] X. Xie, Y. Liang, Y. Wang, G. Sun, and T. Wang, "Coordinated static and dynamic cache bypassing for GPUs," in *Proc. IEEE 21st Int. Symp. High Perform. Comput. Archit. (HPCA)*, Burlingame, CA, USA, Feb. 2015, pp. 76–88.

[29] X. Xie *et al.*, "Enabling coordinated register allocation and thread-level parallelism optimization for GPUs," in *Proc. 48th IEEE/ACM Annu. Int. Symp. Microarch. (MICRO)*, Honolulu, HI, USA, Dec. 2015.

[30] Y. Liang, Z. Cui, K. Rupnow, and D. Chen, "Register and thread structure optimization for GPUs," in *Proc. 18th Asia South Pac. Design Autom. Conf. (ASP-DAC)*, Yokohama, Japan, Jan. 2013, pp. 461–466.

[31] X. Chen, Y. Wang, Y. Liang, Y. Xie, and H. Yang, "Run-time technique for simultaneous aging and power optimization in GPGPUs," in *Proc. 51st Annu. Design Autom. Conf. (DAC)*, San Francisco, CA, USA, 2014, pp. 1–6.

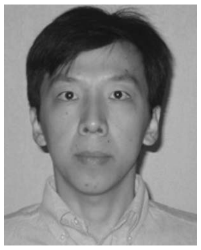
[32] Y. Liang, H. P. Huynh, K. Rupnow, R. S. M. Goh, and D. Chen, "Efficient GPU spatial-temporal multitasking," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 3, pp. 748–760, Mar. 2015.

[33] M. Rhu and M. Erez, "The dual-path execution model for efficient GPU control flow," in *Proc. IEEE 19th Int. Symp. High Perform. Comput. Archit. (HPCA)*, Shenzhen, China, Feb. 2013, pp. 591–602.

[34] A. S. Vaidya, A. Shayesteh, D. H. Woo, R. Saharoy, and M. Azimi, "SIMD divergence optimization through intra-warp compaction," in *Proc. 40th Annu. Int. Symp. Comput. Archit. (ISCA)*, Tel Aviv, Israel, 2013, pp. 368–379.

[35] M. Rhu and M. Erez, "Maximizing SIMD resource utilization in GPGPUs with SIMD lane permutation," in *Proc. 40th Annu. Int. Symp. Comput. Archit. (ISCA)*, Tel Aviv, Israel, 2013, pp. 356–367.

[36] Y. Lee *et al.*, "Exploring the design space of SPMD divergence management on data-parallel architectures," in *Proc. 47th Annu. IEEE/ACM Int. Symp. Microarch. (MICRO)*, Cambridge, U.K., 2014, pp. 101–113.



**Yun Liang** (M'10) received the B.S. degree in software engineering from Tongji University, Shanghai, China, in 2004, and the Ph.D. degree in computer science from the National University of Singapore, Singapore, in 2010.

He was a Research Scientist with the Advanced Digital Science Center, University of Illinois at Urbana–Champaign, Urbana, IL, USA, from 2010 to 2012. He has been an Assistant Professor with the School of Electronics Engineering and Computer Science, Peking University, Beijing, China, since 2012. His current research interests include graphics processing unit architecture and optimization, heterogeneous computing, embedded system, and high-level synthesis.

Dr. Liang was a recipient of the best paper award in International Symposium on Field-Programmable Custom Computing Machines 2011 and the best paper award nominations in International Conference on Hardware/Software Codesign and System Synthesis 2008 and Design Automation Conference 2012. He serves as a Technical Committee Member for Asia South Pacific Design Automation Conference (ASPDAC), Design Automation and Test in Europe, International Conference on Compilers Architecture and Synthesis for Embedded System, and International Conference on Parallel Architectures and Compilation Techniques. He is the TPC Subcommittee Chair for ASPDAC 2013.

**Muhammad Teguh Satria** received the B.S. degree in mathematics from the Bandung Institute of Technology, Bandung, Indonesia, in 2006, and the M.S. degree in computer science and information engineering from the National Taipei University of Technology, Taipei, Taiwan, in 2009.

Since 2009, he has been researching the CUDA development projects for geothermal, computer vision, space science, and general computing projects. From 2012 to 2014, he was with the Advanced Digital Sciences Center, Singapore.

**Kyle Rupnow** received the Ph.D. degree in electrical engineering from the University of Wisconsin–Madison, Madison, WI, USA, in 2010.

He is currently a Research Scientist with the Advanced Digital Science Center, University of Illinois at Urbana–Champaign, Urbana, IL, USA, and an Assistant Professor with Nanyang Technological University, Singapore.

Dr. Rupnow was a recipient of the Sandia National Laboratories Excellence in Engineering Fellowship, the NSF Graduate Research Fellowship (honorable mention), the Gerald Holdridge Award for Tutorial Development, and the University of Wisconsin–Madison Capstone Ph.D. Teaching Award.

**Deming Chen** (S'01–M'05) received the B.S. degree from the University of Pittsburgh, Pittsburgh, PA, USA, in 1995, and the M.S. and Ph.D. degrees from the University of California at Los Angeles, Los Angeles, CA, USA, in 2001 and 2005, respectively, all in computer science.

He has been an Associate Professor with the Department of Electronics and Communication Engineering, University of Illinois at Urbana–Champaign, Champaign, IL, USA, since 2011. His current research interests include system-level and high-level synthesis, nano-systems design and nano-centric CAD techniques, graphic processing unit and reconfigurable computing, hardware/software codesign, and computational biology.

Dr. Chen was a recipient of the NSF CAREER Award in 2008, five best paper awards for Asia South Pacific Design Automation Conference 2009, SASP 2009, Field-Programmable Custom Computing Machines 2011, SAAHPC 2011, and the International Conference on Hardware/Software Codesign and System Synthesis 2013, the ACM SIGDA Outstanding New Faculty Award in 2010, and IBM Faculty Award in 2014 and 2015. He is or has been an Associated Editor of the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS, TODAES, the IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS, and the IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS–I: REGULAR PAPERS.