On-GPU Thread-Data Remapping for Branch Divergence Reduction

HUANXIN LIN, CHO-LI WANG, and HONGYUAN LIU, The University of Hong Kong

General Purpose GPU computing (GPGPU) plays an increasingly vital role in high performance computing and other areas like deep learning. However, arising from the SIMD execution model, the branch divergence issue lowers efficiency of conditional branching on GPUs, and hinders the development of GPGPU. To achieve runtime on-the-spot branch divergence reduction, we propose the first on-GPU thread-data remapping scheme. Before kernel launching, our solution inserts codes into GPU kernels immediately before each target branch so as to acquire actual runtime divergence information. GPU software threads can be remapped to datasets multiple times during single kernel execution. We propose two thread-data remapping algorithms that are tailored to the GPU architecture. Effective on two generations of GPUs from both NVIDIA and AMD, our solution achieves speedups up to 2.718 with third-party benchmarks. We also implement three GPGPU frontier benchmarks from areas including computer vision, algorithmic trading and data analytics. They are hindered by more complex divergence coupled with different memory access patterns, and our solution works better than the traditional thread-data remapping scheme in all cases. As a compiler-assisted runtime solution, it can better reduce divergence for divergent applications that gain little acceleration on GPUs for the time being.

CCS Concepts: • **Computing methodologies** → **Vector/streaming algorithms**; • **Software and its engineering** → **Source code generation**; *Preprocessors*;

Additional Key Words and Phrases: Parallel computing, GPGPU, SIMD, branch divergence

ACM Reference format:
Huanxin Lin, Cho-Li Wang, and Hongyuan Liu. 2018. On-GPU Thread-Data Remapping for Branch Divergence Reduction. *ACM Trans. Archit. Code Optim.* 15, 3, Article 39 (September 2018), 24 pages.
https://doi.org/10.1145/3242089

1 INTRODUCTION

These days, Graphics Processing Units (GPUs) are no longer merely dedicated to graphics processing. As energy consumption has become a significant concern in high performance computing, fewer supercomputers are built with multicore processors only. Programming models such as OpenCL [23] and CUDA [17] have facilitated general purpose computing on GPU (GPGPU). On the Top500 [8] list, the current number of GPU supercomputers is 87, which has increased 8 times over the last 7 years. Moreover, GPU has become the most popular platform for deep learning.

This research is supported by Hong Kong RGC GRF 106160098.
Authors' addresses: H. Lin, C.-L. Wang, and H. Liu, Room 414, Chow Yei Ching Building, The University of Hong Kong; emails: {hxlin, clwang, hyliu}@cs.hku.hk.

1544-3566/2018/09-ART39 $15.00
https://doi.org/10.1145/3242089

The GPGPU community has been shedding light on the branch divergence issue [6, 10, 15, 16, 28], which originates from the Single Instruction Multiple Data (SIMD) execution model. In the hardware design, GPU cores are grouped into Compute Units (CU, AMD term) or Streaming Multiprocessors (SM, NVIDIA term). Members of each CU/SM must always execute the same instruction in lockstep. When it comes to conditional branching, each CU/SM sequentially executes all the branch paths taken by the GPU software threads, or *work-items*[1] that it is hosting, and then masks off wasteful computation results. An if-else branch can already halve GPU execution efficiency, and nested branches lower computation performance exponentially.

Nontheless, efforts are never stopped to accelerate applications with complex branches on the GPU platform. The GPGPU frontier has been pushed to highly divergent areas including computer vision [19], algorithmic trading [4], data analytics [3], artificial intelligence [12], and so on. Nested branches with over four layers can be found in some applications [5, 14] that have been ported to GPU for better cost-effectiveness, promising even further acceleration. In such kernels with frequent branching, the majority of branch conditions cannot be evaluated at compile-time. As a case in point, branches may be wrapped in loop and thus be evaluated to different values across iterations. Even if all the conditions could be correctly predicted, each divergent branch in a single GPU kernel may possess different characteristics, demanding multiple rounds of divergence reduction.

On-CPU Thread-Data Remapping (TDR) is by far the most widely used software solution [7, 15, 27, 28], but it fails to cater to the new challenges of highly divergent applications. In the context of branch divergence reduction, each group of work-items, or *wavefront*, is remapped to datasets that produce the same branch condition, so that the hosting CU/SM only needs to execute one branch path. So far, TDR is merely performed as compile-time preprocessing on the host machine that transfers input data and offloads computation as kernels to the GPU device. On-CPU TDR tries to evaluate branch conditions in advance and rearrange the input data accordingly, but the branch conditions often depend on runtime computation results. It causes a lot of redundant computation to get the actual results and then restart the kernel after on-CPU TDR. As a compile-time solution, traditional TDR can only change the thread-data mapping once for each kernel launch. When there are multiple divergent branches in a kernel, an optimal thread-data mapping has to be determined to cover different needs of the branches. However, with each branch demanding a different mapping, on-CPU TDR has to compromise and miss out on some of the speedup opportunities. The current remedy for these two issues is kernel splitting [25]. A feedback-optimization loop is established, where CPU works on the runtime GPU-feedback branch conditions and performs remapping for the following kernel splits. However, this involves great overhead covering kernel launching and round-trip data transfer.

To handle divergent applications flexibly, we propose Workgroup-Autonomous GPU-NativE Reference Redirection (WAGNERR) as a compiler-assisted GPU-runtime solution. Since WAGNERR leverages on-GPU TDR, the long-latency feedback-optimization loop in traditional TDR can be skipped and kernel splitting is not necessary. Our algorithms are designed to run parallelly on the GPU architecture, and aim to preserve the native latency, hiding as much as possible. WAGNERR can realize fine-grained branch divergence reduction, which has a twofold meaning. First, a kernel is not the smallest unit for divergence reduction as in on-CPU TDR. Multiple branches in each kernel are treated separately to resolve conflict of demanded mappings. Immediately before each branch, one pass of TDR is performed according to actual branch conditions, and then execution is resumed right at the branch with minimized redundant computation. Second, each workgroup is autonomous and responsible for the TDR of member work-items. Logical work-item groupings and shared memory usage will not be broken as in some cases of on-CPU TDR [15].

[1] In this article, GPU software entities are referred to in OpenCL terms. "Thread" and "work-item" are used interchangeably.

Without any centralized decision-making on the new thread-data mapping, execution synchronization is not imposed on workgroups, leaving inter-workgroup context switching unimpaired.

The contributions of this work are:

- To achieve fine-grained branch divergence reduction, we establish WAGNERR as the first on-GPU TDR scheme. It only requires source-to-source transformation on the GPU kernels, which can be widely applied and automated using code parsers.
- We propose *Head-or-Tail* as an on-GPU TDR algorithm that is specialized for the most common two-path branches. It minimizes long-latency global memory accesses for TDR computation and incurs an O(m) overhead, where m is the number of global memory arrays accessed for branch condition computation.
- We propose *Data Group Indexing* (DGI) as another on-GPU TDR algorithm that works for general many-path branches. It features a sub-count index on branch conditions that narrows down the per-thread search space. With adjustable overhead on GPU shared memory, it enhances the applicability of our solution. Global memory accesses are cut to O(m∗log n), where n is the kernel workgroup size.

By realizing runtime on-GPU TDR, our solution can accelerate branches with compile-time unknown conditions. Fine-grained divergence reduction is achieved without kernel splitting. It is no longer a must to compute one and only one thread-data mapping for multiple branches in a kernel, which changes the paradigm of TDR in divergence reduction.

OpenCL benchmarks are tested on two generations of both NVIDIA and AMD GPUs. Experiments show that WAGNERR achieves on average 71% of the potential performance improvement indicated by the profiled control flow efficiency for third-party applications, with the largest achieved speedup of 2.718. For detailed evaluation, we implemented three GPGPU-frontier benchmarks, and WAGNERR proves to be the better solution compared with the traditional on-CPU TDR.

The rest of this article is organized as follows. Section 2 features background knowledge on TDR and the design challenges of our solution. Section 3 shows the details of WAGNERR. Section 4 shows evaluation results. Finally, Section 5 is related work, and last comes the conclusion.

2 BACKGROUND AND DESIGN CHALLENGES

To the best of our knowledge, no previous work has achieved on-GPU TDR. This section reviews background of TDR and reveals design challenges of our solution.

2.1 On-CPU Thread-Data Remapping

In the context of branch divergence reduction, threads in the same wavefront are remapped to data that would produce the same branch conditions or a similar sequence of conditions. In other words, datasets can be divided into *groups* based on which path or path vector they point to. Multiple branches may impose multiple groupings on the datasets, which may conflict with each other and thus demand different ideal thread-data mappings. To achieve complete divergence reduction, data in the same group should be fed to threads in the same wavefront, following each mapping demanded by each branch. In fact, there are two approaches to realize TDR:

Reference Redirection (RR). RR redistributes reference indices to data arrays among threads. In GPU programming, the reference index is often the thread ID (*tid*), or an arithmetic expression of it (e.g., *data[tid*2]*). Therefore, it is adequate to exchange thread IDs, so that data references issued within each wavefront are redirected to data from the same group (e.g., *data[newTid]*).

Data Layout Transformation (DLT). DLT reorganizes physical layout of data storage. Typically, input data arrays are sorted according to the different data groups. Threads then access the reorganized data with their original reference indices (e.g., *newData[tid]*).

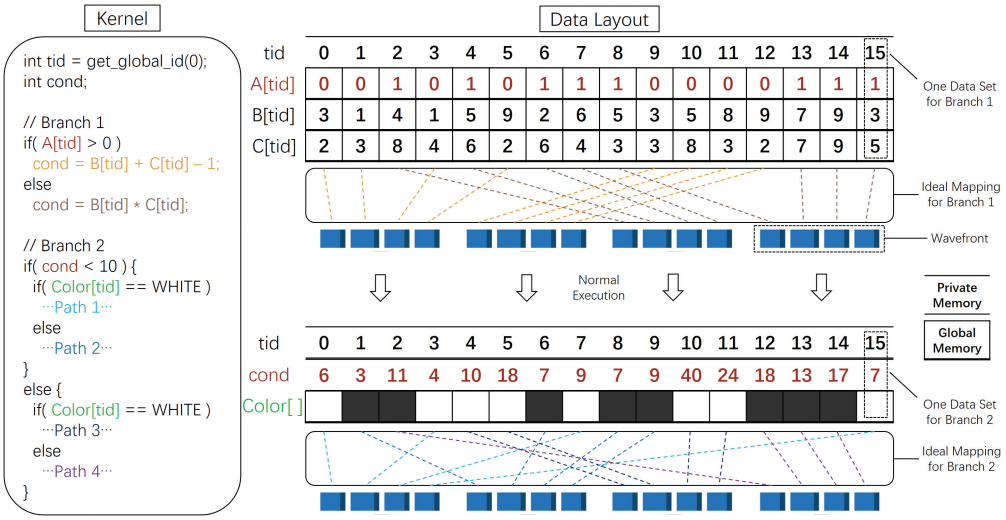


Fig. 1. Example of a GPU kernel with two branches, with Branch 2 dependent on computation results of Branch 1. Data groups are labeled with different colors corresponding to branch paths, and the branches demand different ideal thread-data mappings. Wavefront size is 4.

As pointed out by Zhang et al. [27], these two implementations are of similar computation complexity on CPU. Nevertheless, RR often gives rise to non-consecutive reference indices and thus memory coalescing overheads.

Memory coalescing is a GPU-specific issue. Work-items within a wavefront usually reference consecutive memory addresses that fall into one memory segment perfectly (e.g., *data[tid]*). Only one memory transaction is required for each wavefront. However, when referenced data are scattered across multiple memory segments, multiple memory transactions have to take place to bring in all the referenced memory segments. A designated hardware component will then align the referenced data into one memory segment to be accessed by the wavefront. Overhead comes with the extra memory transactions and hardware coalescing. As a result, Data Layout Transformation is the more widely used implementation for traditional TDR.

Drawback of on-CPU TDR is showcased in Figure 1. In the given kernel, the outer branch condition of Branch 2 is dependent on the resulting variable *cond* of Branch 1, and thus Branch 2 cannot be evaluated at compile-time. Moreover, the two branches access different arrays and impose different groupings on their datasets, which causes the conflict between ideal mappings. From the perspective of RR, each thread should take up different IDs for each of the two branches so as to achieve peak efficiency.

To address the problems, traditional TDR often turns to the heuristic of data sorting [15]. On the host-end, input data are sorted numerically, assuming that similar numerical value will produce the same branch value. However, such an assumption does not always hold, and when there are multiple data arrays as in Figure 1, it is unclear which array should be the sorting key. In this article, we will demonstrate why on-GPU TDR is a more elegant solution achieving fine-grained divergence reduction.

2.2 TDR on GPU Memory Hierarchy

GPU has a three-tier memory hierarchy that is different from CPU, which is critical to TDR.

Private Memory. Each work-item possesses a piece of the fastest exclusive private memory.

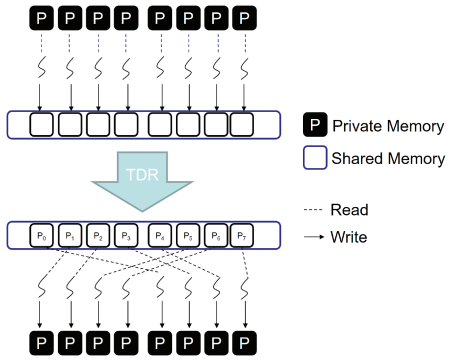


Fig. 2. Private data are put onto shared memory, and then retrieved by new owner after on-GPU TDR.

Shared Memory. Each workgroup has shared memory that can be accessed and synchronized by its members.

Global Memory. The largest and slowest piece of memory is global memory, which can be accessed by all work-items. However, since workgroup-level execution synchronization is too costly, global memory data modifications are often not visible to all work-items.

After TDR, datasets need to be accessible to their new owner. However, correct exchange of data cannot be guaranteed if two exchanging threads come from different workgroups, because in that case global memory is the only layer that is accessible to both parties.

Therefore, on-GPU TDR should be performed via shared memory and thus within workgroups to maximize efficiency. As shown in Figure 2, each work-item can write its private data onto shared memory before TDR (either RR or DLT), which are retrieved by the new owner afterward. The data in shared memory and global memory are already available to peers in the same workgroup. The necessary communications for TDR computation can also be made via shared memory.

In a nutshell, on-GPU TDR is feasible within individual workgroups. As pointed out by Liang et al. [15], on-CPU TDR is also often limited to the intra-workgroup scale due to use of shared memory and logical groupings. On-GPU TDR algorithms need to be carefully designed so that the overheads in private and shared memory will not lead to drop in GPU occupancy.

2.3 DLT vs. RR on GPU

The two TDR approaches ought to be compared in the new context of GPU runtime. Numbers in the following discussion are obtained from prototype tests done for both approaches.

The tests are conducted on NVIDIA GTX 980 with the micro-benchmark as shown in Listing 1. The main body of the kernel is a short if-else branch. Input array *opd* stores the operand, and *bc* stores the branch condition for each thread. Global memory accesses are minimized to the initial reads and the final write. To weaken the latency hiding effect arising from GPU context switches, the total number of threads is set to a small value (1,024). With such settings, we let TDR overhead dominate the kernel execution time. Branch conditions in *bc* are either 0 or 1 with equal probabilities. Outputs are checked against a CPU version.

Data Layout Transformation. DLT needs to move data in the physical storage, and the updated layout ought to be accessible to all work-items. Unfortunately, in most cases, branch-related data (input arrays) are stored in the slowest global memory, where modifications are not guaranteed visible to all. Therefore, global memory data should also be copied to shared memory. Experiments

```
1  __kernel void simple_branch(__global int *bc,
       __global int *opd)
2  {
3      int glid = get_global_id(0);
4      int op = opd[glid];
5      int cond = bc[glid];
6
7      if (cond > 0) {
8        op = (op + 15);
9        op = (op * op);
10       op = ((op * 2) - 225);
11     } else {
12       op = (op * 2);
13       op = (op + 30);
14       op = (op * (op - 15));
15     }
16
17     opd[glid]= op;
18 }
```

Listing 1. Micro-Benchmark Used in Prototype Tests.

show that if each work-item copies one integer from global memory to shared memory in a coalesced fashion, an overhead of 0.1ms will be incurred.

Reference Redirection. RR only requires the computation of new thread IDs, and thus needs few data movements. Prototype tests show that the overhead is around 0.01ms including TDR computation and worst-case memory coalescing.

For on-GPU TDR, memory coalescing overhead seems much smaller than that of global memory data movement. In fact, intra-workgroup TDR puts an upper-bound on the number of required memory transactions for each wavefront. After TDR, member threads will not access data owned by another workgroup. Thus from the perspective of a workgroup, the referenced data remain unchanged, so no extra memory transaction is needed. Wavefronts in the same workgroup may benefit from the caching effect, and thus the major overhead of RR is incurred by hardware coalescing, which has been optimized across GPU generations. Our further study confirms that RR is advantageous over DLT on GPU.

- —RR requries only one pass of computation, because it can be realized by exchanging thread IDs. Each thread is associated with only one ID, but it may need to access multiple arrays, which leads to multiple rounds of data movements in DLT.
- —DLT may also lead to non-coalesced memory accesses, which are found in common GPU sorting algorithms such as Bitonic sort [20]. In addition, such algorithms run passes of swapping on shared memory, which requires plenty of synchronization. We also tested a prototype for DLT using Bitonic sort, and the overhead is around 0.5ms.
- —RR allows better parallelism and decentralization. First of all, the need for central storage of mapping information vanishes. New ID of one thread is worthless to another. Since TDR becomes part of the kernel, there is also no need to transfer mapping information to another device (e.g., from CPU to GPU in traditional TDR). On the other hand, the RR-based TDR computation itself can be fully parallelized and decentralized. Each thread can compute its new ID without causing data races.

Therefore, RR is adopted in our solution.

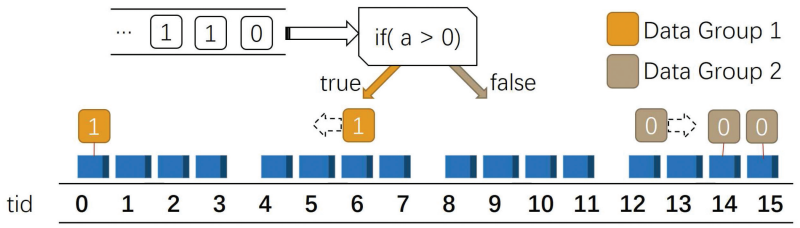


Fig. 3. On-the-fly mechanism specialized for two-path branches. Datasets are inspected individually and then remapped to one end of the work-item space, depending on the branch value. Parallelization is enabled by GPU atomic instructions.

2.4 Correctness of Reference Redirection in Kernels with Dynamic Thread IDs

RR makes changes to thread IDs, which may lead to a worry that it breaks correctness for kernels that are programmed to change thread IDs dynamically. This subsection is dedicated to relieving such concerns.

Such kernels consist of multiple phases or iterations, each corresponding to a different task. At the beginning of a new phase, threads obtain a new ID using atomic operation. This technique is widely used for primitives like prefix scan. One of our evaluation benchmarks, *nqueens* from OpenDwarfs [14], also utilizes dynamic thread IDs.

In both this technique and RR, the system-assigned ID is never changed. The changed entity is actually a private memory variable that stores the current ID value. When the ID variable is updated, the reference indices in the new phase are based on this new ID value, because it is what distinguishes a thread from others. The one-to-one mapping between threads and data holds within each phase, so the correctness of RR is ensured within each phase.

Performing RR in the phase that the branch belongs to will not affect the correctness of other phases. Threads obtain new identity normally at the border of phases. It is true that the intermediate results or work progress by a thread in an earlier phase are picked up by a different thread in a later phase. As long as every work progress is picked up and threads do not pick up the same progress, the results will be correct.

In the principle of OpenCL, workgroup is the logical unit for execution and synchronization. The correctness of kernels that properly obey programming interfaces will not be affected by the remapping performed within workgroups.

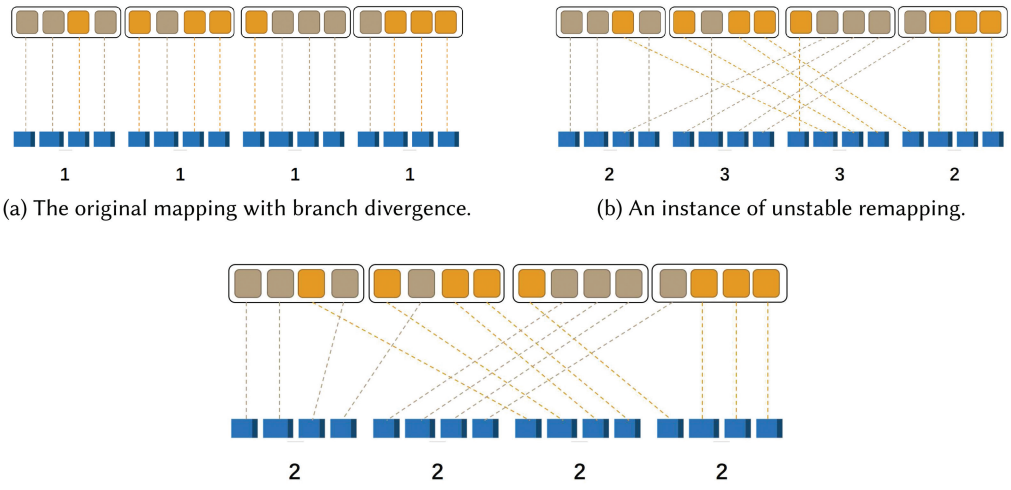
Therefore, RR works in kernels with dynamic thread IDs, and the design of algorithms is critical.

2.5 General Discussion of On-GPU TDR Algorithms

We believe the bottleneck of on-GPU TDR lies in memory accesses, especially because most branch-related data are stored in global memory. The mapping decision depends on the actual data distribution, knowing which may require a round of data traversal. Then, to locate its new set of data, each thread may have to traverse some data again, if not all.

Nontheless, due to the prevalence of if-else syntax, a typical target branch has two divergent paths and thus two data groups. We designed an on-the-fly mechanism that does not require prior knowledge of the number of data belonging to each group, which is illustrated in Figure 3.

In the mechanism, datasets are inspected individually. A Group-1 set will be remapped to thread with the current smallest ID, and a Group-2 set will be remapped to thread with the current largest ID. Upon finishing, threads running the two paths are separated with a border on ID.



(a) The original mapping with branch divergence.

(b) An instance of unstable remapping.

(c) A *stable remapping* is one that maintains original intra-group order of data, assigning nearby data sets to each wavefront.

Fig. 4. Three thread-data mappings, labeled with number of memory segments referenced by each wavefront. Apart from reducing branch divergence, *stable remapping* reduces memory coalescing overhead as a heuristic.

Single-layer branch divergence can be targeted with algorithms that implement this mechanism and handle two paths at a time, referred to as *two-path* algorithms. Such algorithms inspect each dataset only once, which reduces memory accesses optimally.

Although, in principle, two-path algorithms can be applied on nested branches recursively, there is a strong motivation for other designs. Depending on the amount of private data to be exchanged over shared memory, the basic cost for each round of TDR may be expensive. There is a need for algorithms that handle more than two paths in each round, or *many-path* algorithms. Given more than two paths under consideration, it becomes necessary to count each group of data before TDR.

The reason is as follows. Suppose Group-3 data are added into the scenario in Figure 3, and they are to be gathered in a region between the other two groups. Without knowing the numbers of Group-1 and Group-2 datasets, the borders are unknown so the remapping position of Group-3 data is unclear. Every remapping decision on Group-3 data can be proved wrong by future data sequences, scattering Group-3 data among the other two groups. This problem intensifies with the increase of branch paths.

Therefore, in a many-path algorithm, the remapping decision should be preceded by group-wise data counting. The count for each data group is useful not only for making TDR decisions, but also for detecting non-divergent branches. For instance, TDR can be skipped if all data lead to the same path. The counts should be obtained in a way that makes good use of GPU parallelism, and should be known to all threads in the same workgroup.

In the remapping phase that follows the counting phase, we believe it is advantageous to maintain the original intra-group order of data, which we denote as *stable remapping*.

For the example in Figure 4(a), a one-to-one mapping needs to be determined after collecting the branch value of each dataset. As *stable remapping* is shown in Figure 4(c), ascending reference indices to a data group are taken by consecutive threads in the ascending order of ID. Other mappings may achieve the same divergence reduction (Figure 4(b)), but the coalescing overhead is larger.

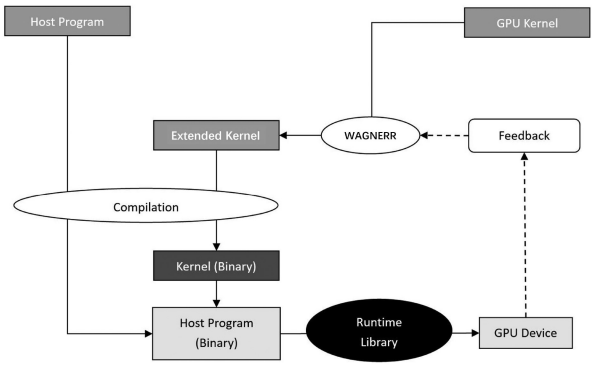


Fig. 5. On the host machine, WAGNERR engine inserts TDR codes into the original GPU kernel, which is then compiled normally. Feedback is handled if the kernel is launched multiple times.

The merit of *stable remapping* is twofold. On one hand, it provides a universal and fully parallel mechanism for each thread to determine a remapping. The uniqueness of thread ID guarantees a unique ranking among the threads and thus a correct one-to-one mapping. On the other, this simple strategy also serves as a heuristic to reduce coalescing overhead. Data referenced by a wavefront will not be sparsely scattered but close to each other, which may come from fewer memory segments.

3 WORKGROUP-AUTONOMOUS GPU-NATIVE REFERENCE REDIRECTION

In this section, we present our solution framework and two on-GPU TDR algorithms.

3.1 Solution Framework

WAGNERR is a compiler-assisted runtime solution framework that performs source-to-source transformation on GPU kernels. As shown in Figure 5, the WAGNERR engine statically analyzes divergence in the original kernel, and generates the extended kernel by inserting on-GPU TDR codes. At runtime, the inserted codes will react to the divergence arising from the actual input data and achieve the best thread-data mapping for each target branch. In the case where the same kernel is launched multiple times in a loop, runtime information can be passed back to adjust algorithm-specific configuration. Workflow of WAGNERR engine mainly includes:

—Identify divergent branches
—Analyze dependencies among branches, and divide the kernel into dependency-free fragments
—Insert on-GPU TDR codes into each branch block
—Check if memory usage leads to drop in GPU occupancy; if yes, adjust algorithm configuration
—Declare variables and shared memory at the beginning of the kernel

WAGNERR minimizes the number of TDR rounds by dividing the kernel into dependency-free fragments. Each fragment contains one or more divergent branches, but their branch conditions only depend on computation in previous fragments. Dependency checking is carried out by locating the last modification of variables involved in branch condition computation. These branches determine the number of branch paths within the fragment, and thus the data groups in each TDR round. After TDR, kernel execution resumes right at the branch without redundant computation.

To prevent drop in GPU occupancy due to on-GPU TDR, WAGNERR is cautious with overheads in registers and shared memory. Both resources are reused in all TDR rounds for the same kernel. Thus, the amount of shared memory to declare at the beginning of the kernel is the largest shared memory usage among the kernel fragments.

More implementation details will be discussed after the presentation of two algorithms based on RR. Please be reminded that the algorithms are fully parallel, and work-item is the unit of execution for the described steps. Instructions including atomic ones are all supported by both OpenCL and CUDA, making WAGNERR portable on state-of-the-art GPUs. Our primary design philosophy is to minimize global memory accesses.

3.2 Head-or-Tail (HoT)

HoT is a two-path algorithm implemented with atomic operations on a pair of shared variables, *head* and *tail*. Initially, they mark the head and tail slots of *idpool*, a shared array where threads exchange their IDs. Depending on its original branch condition, each thread writes its ID to the slot either pointed by *head* or *tail*, which are accessed and updated atomically. Regardless of the distribution of two groups of data, eventually *head* and *tail* cross each other, and *idpool* is full. After synchronization, each thread reads from *idpool* with its old ID as the reference index.

ALGORITHM 1: Head-or-Tail (HoT)

Input: workgroup size S, main data array $data[S]$, and thread ID tid
Output: new ID for each thread $newtid$

```
1:  Define idpool[S], head and tail in shared memory
2:  head ← 0
3:  tail ← S − 1
4:  Intra-workgroup synchronization
5:  Compute branch condition cond from data[tid]
6:  if cond == true then
7:      val ← tid
8:      slot ← atomic_add(&head, 1)
9:  else
10:     val ← tid − S
11:     slot ← atomic_sub(&tail, 1)
12: end if
13: idpool[slot] ← val
14: Intra-workgroup synchronization
15: newtid ← idpool[tid]
16: if newtid is out of range then
17:     newtid ← newtid + S
18:     cond ← false
19: else
20:     cond ← true
21: end if
```

Detail steps are listed in Algorithm 1. For presentation simplicity, there is only one data array under consideration. In reality, dataset for branch condition computation may be more complex, and the index to data arrays may not always be *tid*. Such information can be obtained with static analysis, and the inserted codes can be adjusted accordingly. This applies to *DGI* as well.

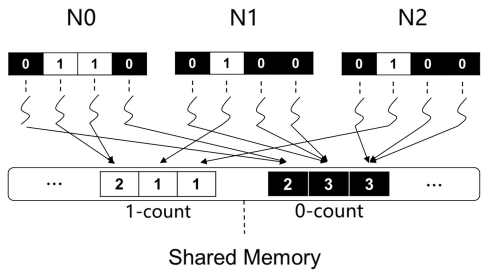


Fig. 6. An illustrative example of two-level parallel counting in *DGI*. Sub-counts are collected by three neighborhoods.

First, each thread reads its original dataset and computes the branch condition *cond* (Line 5). If *cond* is true, thread will read the value of *head* and write its thread ID (*tid*) into the corresponding slot (Line 13). If *cond* is false, thread will access *tail*, and the value written to that slot is *tid* minus *S* (Line 10).

The two shared variables are accessed and managed by atomic instructions, which perform atomic modification and return the previous value of the variable. Value of *head* grows upward (Line 8), and *tail* grows downward (Line 11). Regardless of how many data are from each group, *head* minus *tail* will always be 1 at the completion of the algorithm.

After synchronization, each thread reads *idpool*[*tid*] (Line 15). There is no re-evaluation of branch condition, as it can be told by whether the read value is within the range of thread ID for this workgroup. If the value is invalid, the new ID will be that value plus *S* (Line 17).

In *HoT*, each dataset is only accessed once for TDR, minimizing global memory accesses.

3.3 Data Group Indexing (DGI)

DGI views threads with consecutive IDs as *neighborhoods*, and their original datasets are called *blocks*. In the first phase, each neighborhood gathers the group-wise data counts of its block, and the results are shared via shared memory. With such information, each thread first determines which group its new data should belong to, and its own ranking adhering to *stable remapping*. Then the shared sub-count array is used as an index to quickly locate the block that carries the data with the correponding rank. From the perspective of each thread, full data traversal is avoided as the search space is narrowed down to one and only one block. Detail steps are listed in Algorithm 2.

3.3.1 Counting Phase. Two-level parallel counting is adopted as shown in Figure 6. For simplicity, the one array is directly labeled with branch conditions, distinguishing two data groups. N0, N1, and N2 are three neighborhood-block pairs.

To collect group-wise data counts from each block, each neighborhood shares a set of *P* counters, where *P* is the number of divergent paths and also the number of data groups. For better locality in later computation, counters for the same data group are located consecutively in the storage layout (e.g., 1-count and 0-count in Figure 6).

In the counting phase, only one data access is made by each thread (Line 2 in Algorithm 2). Each neighborhood evaluates the branch conditions for the block, and uses atomic operation to increment the counters accordingly (Line 4). After intra-workgroup synchronization, group-wise total counts are obtained by each work-item, summing up the counters individually to avoid extra synchronization (Line 6).

ALGORITHM 2: Data Group Indexing (DGI)

Input: workgroup size S, main data array $data[S]$, thread ID tid, number of paths P and neighboring factor N
Output: new ID for each thread $newtid$

1: Define array $counter[P * S/N]$ in shared memory and initialize to 0
2: Evaluate branch condition with $data[tid]$ and determine the path p
3: Neighborhood ID: $nid \leftarrow tid/N$
4: $atomic\_inc(counter + p * S/N + nid)$
5: Intra-workgroup synchronization
6: Every thread computes the number of sets in each data group $\{n_1, n_2, \ldots, n_{p-1}\}$
7: Obtain $nativeid$ using system function
8: **if** $nativeid < n_1$ **then**
9: $newgroup \leftarrow 1$
10: $rank \leftarrow tid$
11: **else if** $nativeid < n_1 + n_2$ **then**
12: $newgroup \leftarrow 2$
13: $rank \leftarrow nativeid - n_1$
14: ...
15: **else**
16: $newgroup \leftarrow P$
17: $rank \leftarrow nativeid - n_1 - n_2 - \cdots - n_{P-1}$
18: **end if**
19: $mark, blockid \leftarrow 0$
20: **while** $rank \geq mark$ **do**
21: $mark \leftarrow mark + counters[newgroup * S/N + blockid]$
22: $blockid \leftarrow blockid + 1$
23: **end while**
24: $newtid \leftarrow min(blockid * N - 1, S - 1)$
25: **while** $mark \neq rank$ **do**
26: Determine the group g where $data[newtid]$ belongs
27: **if** $g = newgroup$ **then**
28: $mark \leftarrow mark - 1$
29: **end if**
30: $newtid \leftarrow newtid - 1$
31: **end while**
32: $newtid \leftarrow newtid + 1$

3.3.2 Remapping Phase.

DGI Implements *Stable Remapping*. Although RR exchanges IDs among threads, the system-generated ID for each thread is never modified, which we denote as *native ID* (Line 7). Threads in the same wavefront have consecutive native IDs, and we do not assume the thread ID is equal to the native one at the start of the algorithm.

From the first $P - 1$ group-wise total counts, *DGI* determines P ranges of consective native IDs corresponding to the data groups. The starting ID of each range is defined in Equation (1):

$$ID(p) = \begin{cases} ID(p-1) + n_{p-1} & p \geq 2 \\ 0 & p = 1 \end{cases}. \tag{1}$$

Each thread compares its native ID with the starting IDs, and determines its *target group*, the data group that it should be remapped to. Its *rank* among threads in the same range is computed as native ID minus starting ID. For the example in Figure 6, the total 1-count is 4. Threads with native IDs smaller than 4 will be remapped to branch condition 1, and the others to 0.

Then, each thread looks up the sub-counts for its target group until the running sum (stored as *mark*) exceeds *rank* (Lines 18–21). The block that it stops at must contain its target dataset.

For example, the thread whose native ID is 7 should be remapped to the fourth dataset that gives branch condition 0. From the sub-count of N1, we know that the block only contains two target-group data, so it should be removed from the search space.

Eventually each thread locates the block that carries its target dataset, and traverses it with descending iterators. Every time it encounters a target-group dataset, it decrements *mark*. When *mark* equals *rank*, new data and ID for the thread are determined.

Thanks to the indexing design, the data search space is narrowed down to one block (Lines 25–31).

3.3.3 Neighboring Factor. Neighboring factor N represents the number of threads in each neighborhood, and there is a tradeoff regarding its value.

Generally speaking, a smaller N requires more shared memory, but leads to less conflicts of atomic increments and a narrower search space. Counters become more fine-grained and provide more information to support the detection of non-divergent wavefronts.

Larger N brings the opposite effects. For example, in the extreme case where N equals work-group size, there is only one counter for each data group. Although total counts are directly obtained, conflicts of atomic increments are maximized, and the search space is never narrowed down.

3.4 Implementation Details

WAGNERR engine is essentially a code parser. Since OpenCL uses subset of C language syntax, we generated a pragma-supported parser with ANTLR [18] using context-free grammar of C for this work. The engine may also be directly incorporated into GPU compilers.

Our engine by default omits branches that depend on thread ID only. Since GPU programmers have been avoiding branch usage, available applications are usually non-divergent [15]. The most common branch is the one that checks whether the thread ID is out of the problem space, which is not truely divergent because the ID is consecutive within each wavefront. Pragmas are available for the user or profiling system to assist divergence identification.

For kernels with target branches, our engine analyzes the current register and shared memory usage. With the input of GPU parameters, WAGNERR computes the budget of registers and shared memory that will not lead to drop in GPU occupancy. On-GPU TDR configurations are adjusted as described below. In the rare case of an extremely-tight budget, the original kernel will be launched.

First, selection of algorithm is based on characteristics summarized in Table 1.

In terms of shared memory, *HoT* requires one integer for each thread ID, and two more for the workgroup (*head* and *tail*). That is a bit higher than four bytes per thread. Since intra-workgroup IDs cover a small range, 16-bit *short* data type is used if the budget on shared memory is tight, cutting usage to two bytes per thread but losing some speed as shared memory access is optimized for 32-bit types. For *DGI*, the overhead is usually smaller than two bytes per thread, as the number of paths is usually smaller than half of the neighboring factor. *DGI* cannot use 16-bit type for shared counters because *short* does not support atomic instructions. Last but not least, variables like array indices that are computed from ID without global memory accesses do not need to be shared explicitly.

Table 1. Characteristics of Two On-GPU TDR Algorithms

Algorithm	*HoT*	*DGI*
Many-Path?	No	Yes
Critical Memory Accesses[2]	O(m)	O(m∗log n)
Shared Memory Overhead[3]	O(n)	O(p∗log n)
Largest Register Overhead per Thread (Bytes)	4	4+4p

In terms of registers, *HoT* requires one to store the intra-workgroup ID, if there is not one already. For *DGI*, we sacrifice readability and explicitly reuse variables as much as possible. The peak register usage is for counting threads going onto each path. For p paths, we compute the first $p-1$ counts that are actually used in the remapping decision, where two more registers are used for storing the new data group for each thread and its rank. Although compilers may further optimize by reusing registers in the original kernel, our engine adds these maximum numbers to compute new register usage for safety. WAGNERR supports the intake of runtime statistics especially related to actual occupancy to further optimize.

Adhering to the strategy of minimizing TDR rounds, our engine will initially select *HoT* for two-path kernel fragments, and *DGI* with 16 as neighboring factor otherwise. WAGNERR makes two-way adjustments until the budget is met. If there is register shortage, the number of paths for each round of *DGI* is reduced or the algorithm is eventually switched to *HoT*. If shared memory is not enough, our engine locates the bottleneck round, and then adjusts by changing the data type for *HoT*, switching from *HoT* to *DGI*, or doubling the neighboring factor of *DGI*. A TDR round is cancelled if it remains the bottleneck when neighboring factor has increased to 64, which is probably due to a large amount of private data to be exchanged. Pragmas are available to overwrite the choice of algorithm.

One more technique is worth mentioning. Since both algorithms use atomic instructions, they can benefit from warp-aggregated atomics [1] if implemented in CUDA, but it is not supported for OpenCL.

4 EVALUATION

In this section, we present the evaluation of our solution on both NVIDIA and AMD GPUs. The speedups on each GPU are computed against the execution time of original kernels on that GPU.

4.1 Methodology

All the tested benchmarks are written in OpenCL so as to run on all four GPU cards listed in Table 2. Below we refer to them with acronyms. They are all hosted by Intel Core i7-4790 CPUs.

Besides speedup and memory statistics, *Control Flow Efficiency* (CFE) is an ideal metric directly assessing divergence severeness. CFE is defined as the percentage of thread instructions that are executed and not masked off due to control flow divergence. Unfortunately, it is in general difficult to collect runtime statistics on both platforms for OpenCL. AMD APP SDK does not provide any branch-related statistics. Although CUDA toolkit comes with an OpenCL profiler, it has been disabled from CUDA 8.0 onward. The remaining CUDA 7.5 profiler does not directly provide CFE for OpenCL, but there is a counter *not_predicated_off_thread_inst_executed* for the N980 card. We

[2] Accesses to condition-related data, which are usually located in global memory. m is the number of input arrays, while n is workgroup size.

[3] p is the number of paths under consideration.

Table 2. GPU Hardware Details

GPU Card	NVIDIA GTX 980	NVIDIA GTX 1080	AMD R9 290X	AMD VEGA 64
Acronym	**N980**	**N1080**	**A9**	**A64**
Wavefront Size	32	32	64	64
Architecture	Maxwell	Pascal	GCN 2.0	GCN 5.0
Register (KB per CU/SM)	512	256	256	256
Shared Memory (KB per CU/SM)	96	96	64	64
Max Resident Threads per CU/SM	2048	2048	2560	2560
GPU Core Count	2048	2560	2816	4096
Peak Clock (MHz)	1216	1733	1000	1546
GFLOPS	4612	8873	5632	12583
Memory Bus (bit)	256	256	512	2048
Memory Clock (MHz)	1753	1607	1250	945
Bandwith (GB/s)	224	320	320	483.8
OpenCL Driver	CUDA 7.5	CUDA 8.0	APP SDK 3.0	APP SDK 3.0
Operating System	Ubuntu 16.04	Ubuntu 16.04	Windows 10	Windows 10

Table 3. Benchmark Information on Resource Usage, Divergence, and Profiled Resource Overheads

Benchmark	BP	HISTO	LBM	PB	SAD	MG	NQ	CVP	GDT	POL
Workgroup Size	192	256	256	256	61	256	256	256	256	256
Register per Thread (Bytes)	20	56	80	28	96	52	44	68	48	64
Shared Memory (SHM) per Thread (Bytes)	0	88	0	0	0	0	0	16	0	4
# of Branch Layers	1	1	1	1	1	2	5	2	3	5
Longest Path (Lines)	1	25	71	1	44	24	174	13	10	27
Global Memory Accesses in Branch	3	4	0	3	2	22	5	9	3	3
CFE on N980	88.6%	95.7%	86.4%	83.4%	94.4%	48.7%	39.4%	25.9%	19.8%	40.1%
Largest Register Overhead (Bytes)	12	8	8	12	4	16	16	16	12	16
SHM Overhead due to Private Data	0	0	0	0	0	0	20	0	16	0
Largest SHM Overhead due to Algorithm	Four bytes per thread (HoT)									

calculate CFE by Equation (2), where 32 is the wavefront size for NVIDIA:

$$CFE = \frac{\text{not\_predicated\_off\_thread\_inst\_executed}}{\text{inst\_executed} \times 32}. \tag{2}$$

Three stages of evaluation are conducted on all four GPUs.

In Stage 1, a micro-benchmark is tested to reveal any systematic flaw in the two TDR algorithms.

In Stage 2, third-party real-life kernels are collected from open-source applications. We use CFE profiling to assist WAGNERR engine locate kernels with branches and actual divergence.

In Stage 3, we implement three divergent benchmarks from the GPGPU frontier (computer vision, algorithmic trading and data analytics), each with a different memory access pattern. Our solution is compared against the traditional on-CPU TDR directly.

Table 3 lists information of benchmarks used in Stages 2 and 3. The listed overheads are the largest value recorded on the four GPUs. Among the benchmarks, only HISTO is forced to a lower

occupancy by large shared memory usage. Thanks to the small overheads, WAGNERR does not lead to occupancy drop in all cases. The information will be revisited in later discussions.

4.2 Micro-Benchmark

The main body of the benchmark kernel is a branch, with a configurable number of paths and minimum codes outside. Branch condition of each thread is directly generated by the host program with equal probabilities. The kernel is compute-intensive, and the divergent paths are permutations of the same set of instructions.

With such an ideal setting, the theoretical slowdown due to divergence equals the number of divergent paths. We apply both on-GPU TDR algorithms to see if they can speed up the execution by this factor. Some key findings are listed below, which are also confirmed in Stages 2 and 3.

HoT. Extra experiments are conducted for *HoT* out of the concern that shared variable *head* and *tail* are atomically accessed by all threads, which has a chance of becoming the bottleneck of TDR.

Inspired by Thread Block Compaction (TBC) [9], we attempt to reduce the number of atomic operations by dividing each workgroup into a few sub-groups and apply *HoT* on each of them. Each thread determines its sub-group by the modulo of its native ID. Similar to TBC, such a design does not guarantee the ideal thread-data mapping.

We tested *HoT* and the sub-group design with various modulo bases, but the atomic operation overhead seemed dominated by other overhead (e.g., synchronization) and the TDR benefits. With a two-path branch, *HoT* achieved a stable speedup of 1.997, while the speedup of the sub-group design was highly fluctuating around 1.5. It seems safe to conclude that *HoT* suits state-of-the-art GPU architecture well, let alone that it will benefit from warp-aggregated atomics in CUDA.

DGI. In various cases, neighboring factor N between 4 and 64 gives similar speedups and CFE. Such a range enables more flexibility in adjusting shared memory overhead. Empirically, 16 is a good starting point for performance tuning, which is also the default value in our current WAGNERR engine. If a kernel is launched multiple times, the sub-count array can be output to the host, so as to assist dynamic adjustment of N.

CFE. CFE proves to be a good estimator of performance at least for compute-intensive kernels. Theoretical speedup is achieved with both algorithms. For an if-else branch, average speedup is 1.995, and CFE rises from 50.0% to 99.7%. For a two-layer nested branch, average speedup is 3.994, and CFE rises from 25.0% to 99.8%.

If branch divergence is completely eliminated and, given the same total workload, CFE will approach 100% as no instruction is wasteful. In fact, software solutions may never totally eliminate real-life divergence due to architectural factors. For example, as long as the number of work-items running one of the paths is not divisible by the wavefront size, there must still be a divergent wavefront after TDR. However, CFE still reflects the proportion and severeness of divergence in a kernel. In Stages 2 and 3, we estimate the potential speedup as 1 over CFE of the original kernel.

4.3 Third-Party Benchmarks

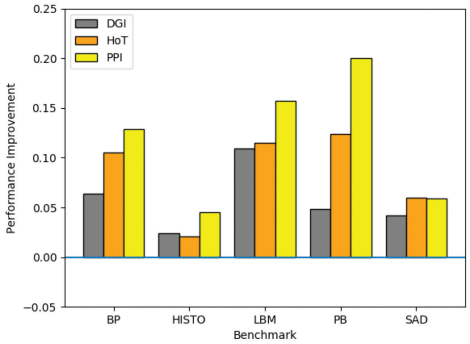
Here we present five benchmarks with single-layer branches, and two with multi-layer ones.

4.3.1 Single-Layer Divergence. The five benchmarks with single-layer branches are:

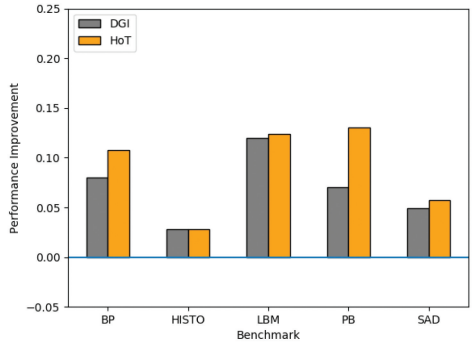
—BackProjection (BP) from the BEMAP benchmark suite [2]
—PoolingBackward (PB) from Caffe [13]
—HISTO, LBM, SAD from the Parboil benchmark suite [24]

Table 4. CFE and Speedups of Third-Party Single-Layer Divergence on N980

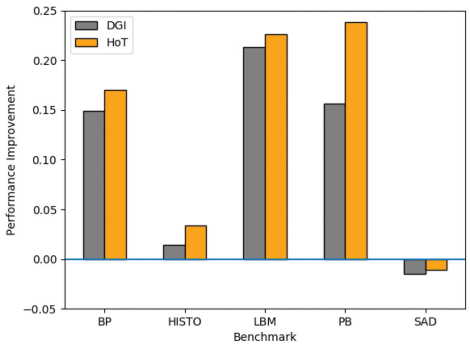
Benchmark	BP	HISTO	LBM	PB	SAD
Original CFE	88.6%	95.7%	86.4%	83.4%	94.4%
Potential Speedup (1/CFE)	1.129	1.045	1.157	1.200	1.059
CFE after DGI	95.3%	98.0%	95.8%	88.3%	97.5%
Speedup after DGI	1.064	1.024	1.109	1.048	1.042
CFE after HoT	98.2%	98.9%	96.3%	94.4%	98.4%
Speedup after HoT	1.105	1.021	1.115	1.124	1.060



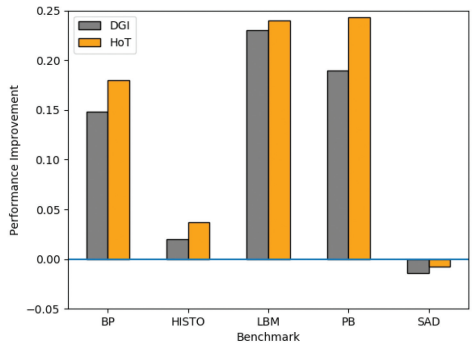
(a) NVIDIA GTX 980 (PPI stands for Potential PI)



(b) NVIDIA GTX 1080



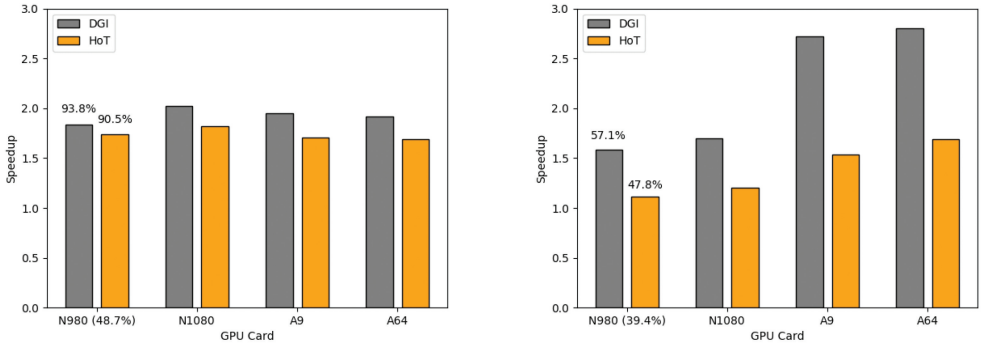
(c) AMD R9 290X



(d) AMD VEGA 64

Fig. 7. Performance improvement (speedup minus 1) on third-party benchmarks with single-layer branches.

The results are displayed in Table 4 and Figure 7. As expected, *HoT* is more effective with single-layer branches, giving higher speedup and CFE. To evaluate the completeness of divergence elimination, we use performance improvement (PI, speedup minus 1) to avoid being mislead by the non-negativity of speedup. As results on N980 are plotted in Figure 7(a), *DGI* averages 49% of the potential PI, while *HoT* averages 74%. With at most two paths under consideration, *HoT* is a better option given that it minimizes global memory accesses. Differences in the final CFE for the two algorithms can be explained by the fact that more branches are used in the steps of *DGI*. In



(a) Speedups of MG on 4 GPUs (b) Speedups of NQ on 4 GPUs

Fig. 8. Third-party multi-layer divergence under WAGNERR's impact. Percentage is CFE, with the baseline number next to the label of N980.

a way, TDR is migrating divergence from long branches to the short ones involved in mapping calculation.

Individual benchmarks are introduced as follows.

BP. BackProjection deals with recovering an image using the projection parameters.

HISTO. Histogram accumulates the number of occurrences in the input for each output value.

The only case where *HoT* is slower than *DGI* happens for HISTO on N980, which may root in the frequent accesses to shared memory. *HoT* incurs a bigger overhead under heavy contention.

LBM. Lattice-Boltzman Method is a partial differential equation solver in fluid dynamics.

PB. PoolingBackward is the backward kernel of pooling layer in the DNN model of deep learning.

SAD. SAD stands for Sum of Absolute Differences, which is used in video compressing to determine similarity between video frames.

This kernel features a workgroup size of 61. Since AMD has a wavefront size of 64, no matter how threads and data are remapped, the sole wavefront in each workgroup will still be divergent. This is an extreme case where our solution should not be considered. As shown in Figure 7(c) and (d), the performance loss is less than 2% on both AMD GPUs.

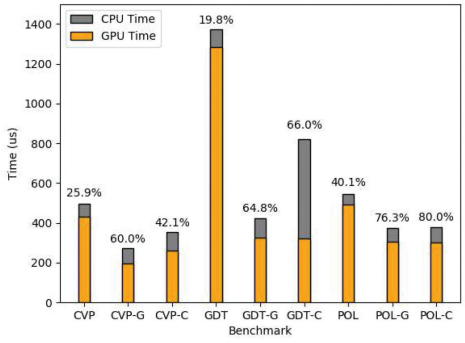
4.3.2 Multi-Layer Divergence. MG from OpenCL NPB [22] and *nqueens* (NQ) from OpenDwarfs [14] are found with multi-layer divergence.

Results are shown in Figure 8. When it comes to branches with many paths, *DGI* conserves TDR rounds and thus incurs smaller overhead in general. The two benchmarks are introduced below.

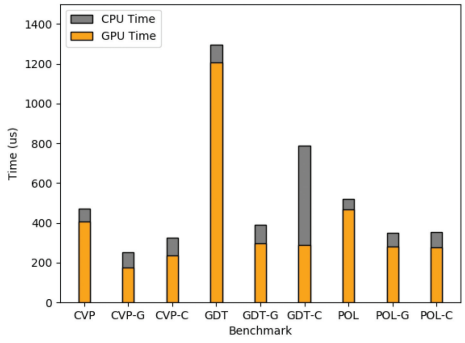
MG. MG is a Multi-Grid solver for computing a three-dimensional potential field. Speedups greater than 1.5 are achieved in all cases, and CFE on N980 is improved to over 90% from 48.7%.

NQ. This N-Queens problem solver has the most complex and divergent OpenCL kernel that we found. Threads look for solutions in parallel, and a new ID is fetched atomically when one searching task is done. Due to the relatively low solution density in the total search space, divergence is less severe than it seems, and the measured CFE on N980 is 39.4%. On that card, the computation speed is estimated to be around 1500 GFLOPS, which is more efficient than running on CPU.

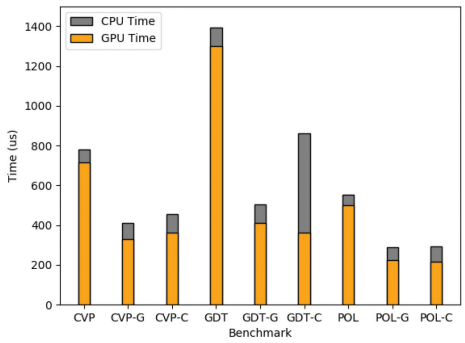
NQ has a giant five-layer branch. Unlike MG, there are computation dependencies between branches in NQ, which requires more rounds of TDR. However, the kernel keeps many private data, which need to be exchanged via shared memory and thus make each round of TDR expensive.



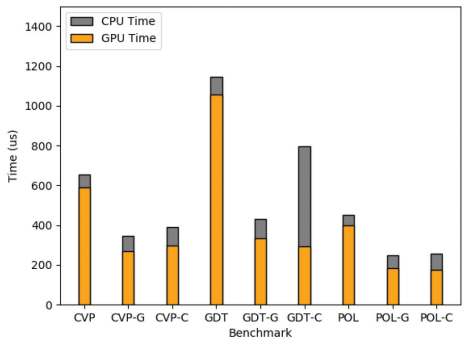
(a) Time on NVIDIA GTX 980 (Percentage is CFE)



(b) Time on NVIDIA GTX 1080



(c) Time on AMD R9 290X



(d) Time on AMD VEGA 64

Fig. 9. Execution time of GPGPU frontier benchmarks on 4 GPUs. Version G uses on-GPU TDR (WAGNERR), while Version C uses on-CPU TDR.

Given that some paths are rarely taken, we manually disabled TDR for the innermost layer, which boosts performance improvement by an average factor of 1.5. This is partially why the final CFE falls relatively short from 100%.

4.4 Divergent GPGPU Frontier Benchmarks

In Stage 3, we implemented GPGPU frontier benchmarks representing three areas, including computer vision, algorithmic trading, and data analytics. They display different memory access patterns. Every benchmark has a baseline version, a WAGNERR-processed version, and an on-CPU TDR (DLT) version.

The results are shown in Figure 9. Combining GPU and CPU time, on-GPU TDR is more effective than the traditional solution in all cases. WAGNERR also incurs a smaller CPU overhead, especially for memory-intensive applications, because on-CPU TDR has to process and rearrange input data every time. Since these kernels all have complex branch nests, WAGNERR selects *DGI* for them, which has been confirmed to be the best setting. Individual benchmarks are examined below.

4.4.1 CVP. CVP is a prediction module for computer vision [19] with a baseline CFE of 25.9%. It represents a memory access pattern where a small amount of data (a tree structure) are frequently accessed by all the work-items.

Due to such an access pattern and compile-time unknown branch conditions, on-CPU TDR can only handle the outmost layer of branch divergence, resulting in longer GPU time than the WAGNERR-processed version.

Final average speedup is 1.9 for on-GPU TDR (CFE 60.0%), and 1.6 for on-CPU TDR (CFE 42.1%).

4.4.2 GDT. GDT is a genetic-programmed decision tree for financial forecasting and algorithmic trading [4] with a baseline CFE of 19.8%. In the GPU kernel, every work-item corresponds to a huge amount of daily data collected from a two-year period. Therefore, GDT represents the big-data access pattern with a small mixture of non-coalesced memory accesses.

Fortunately, for on-CPU TDR, the sorting heuristic can be applied because we are able to identify a major sorting key among hundreds of different data items. The resulting thread-data mapping is highly similar to the one realized by on-GPU TDR. However, rearranging data on CPU takes a very long time even with eight computing threads. For GDT, on-CPU TDR overhead will be difficult to hide.

Final average speedup is 3.0 for on-GPU TDR (CFE 64.8%), and 1.6 for on-CPU TDR (CFE 66.0%).

4.4.3 POL. POL is an original application that assists tax policy decision with a baseline CFE of 40.1%. With a database on tax-related information from thousands of households, POL reveals how tax statistics react to a given change in tax policy. It represents the current mainstream memory access pattern, with coalesced accesses to a medium volume of data.

The sorting heuristic is applied again for on-CPU TDR, and works well to our surprise. It can be explained by the problem nature, because certain patterns exist for households from different social classes. The resulting thread-data mapping is also similar to the one realized by on-GPU TDR.

Final average speedup is 1.7 for on-GPU TDR (CFE 76.3%), and 1.6 for on-CPU TDR (CFE 80.0%).

4.5 Analysis on GPU Architectural Factors

Here we present some analysis on architectural factors with additional statistics.

4.5.1 Slow Global Memory. The existence of global memory access is a good static indicator of severe divergence. Branch paths in BP and PB are very short (1 line), but the global memory accesses amplify the divergence, as they cost thousands of clock cycles.

4.5.2 Impact on Memory Performance. Although RR is notorious for causing non-coalesced memory accesses, it is confirmed that on-GPU TDR actually has a postive overall impact on memory performance. Divergent branch paths have different execution logics and issue different memory accesses. By bringing threads running the same paths together, wavefronts run fewer paths and thus issue fewer memory access. For the nine benchmarks with global memory accesses, L2-cache misses are reduced by 8% (70 misses per 1,000 accesses) on average on the cards with profilers (N980, A9, and A64).

4.5.3 Time Overhead of On-GPU TDR. For GDT and POL, both on-GPU and on-CPU TDR result in similar thread-data mappings, which gives us a better sense of on-GPU TDR time overhead. Since the on-CPU TDR version also comes with the memory benefit, we estimate on-GPU TDR time overhead as their difference in GPU time. It comprises TDR computation and memory coalescing. Instructions introduced by on-GPU TDR rarely involve global memory, so they can be well hidden by the fast native GPU context switching. As for memory coalescing, it is limited to the intra-workgroup scale and optimized in each GPU generation. Therefore, we believe the time overhead should be fairly stable, and will not exceed the value in the case of GDT given that GDT has a huge data volume. Conservatively speaking, the overhead should be less than 0.05ms for any

round of on-GPU TDR on state-of-the-art GPUs. This serves as a small threshold for path execution time, which confirms that branches with global memory accesses are likely to benefit from on-GPU TDR.

4.5.4 GPU Generation. It can be observed that on-GPU TDR works better for newer generations. In hardware development, GPU vendors generally optimize memory coalescing, shared memory, atomic instruction, and so on. The overhead of on-GPU TDR is thus reduced in general.

However, on-GPU TDR speedups are not improved much from A9 to A64 for HISTO, LBM, MG, and GDT, which are all memory-intensive applications. As shown in Table 2, A64 has a higher memory bandwidth, but its memory clock is actually slower. For other benchmarks with smaller data volume, A64 does not have a clear advantage in memory accesses. When it comes to memory-intensive applications, the advantage in memory bandwidth cuts down memory access time, so the TDR overhead is not hidden as well as on A9.

4.5.5 NVIDIA vs. AMD. TDR speedup is generally higher on the AMD platform, and there is an especially huge difference for NQ.

We believe the main reason is the difference in wavefront size. Given the same kernel and data input, CFE is always lower on AMD, and thus promises higher speedups. Consider a workgroup of 256 threads, where the 2 work-items at the two ends take a different path from the other 254. On NVIDIA, 25% of the wavefronts are divergent, but on AMD the percentage is 50%. This effect is amplified by a long rarely taken path in NQ that leads to a lot of masked-off instructions.

Readers may be aware that the CU/SM width is 16 on both platforms as discussed in [26]. Every instruction is actually executed four times on AMD or twice on NVIDIA for the wavefront-splits. However, that does not mean only divergent 16-wide splits need to execute multiple branch paths. Due to pipelining concerns, extra paths are still executed by non-divergent splits. This has been verified with a micro-benchmark, and is also supported by the result of SAD on A9 and A64 in Stage 2.

4.5.6 Impact on Energy Consumption. GPU boost mode is not activated for the experiments, so the power is held steady. In addition, a drop in bank conflicts is observed, implying less activity by the memory components. As a pure software solution, on-GPU TDR reduces computation wastage and shortens the kernel execution time, which helps reduce energy consumption as well.

5 RELATED WORK

Branch divergence solutions for GPU can be classified into two categories: hardware and software.

Fung et al. [10] were the first to address this problem, and laid the foundation for hardware approaches. A series of hardware solutions followed their idea of compacting work-items running the same path into new wavefronts [9, 21, 26]. Hardware solutions can reduce branch divergence with smaller overhead in time, but the integration of new hardware components may affect energy consumption and computation pipeline, which cannot be turned off for non-divergent kernels.

Zhang et al. [27] started to systematically perform on-CPU TDR. They proposed to build a CPU-GPU pipeline that hides the TDR overheads. In their later work [28], they also considered the problem of irregular memory accesses, and designed compile-time methods for computing the best thread-data mapping.

We did not fully re-implement their work because traditional sorting practices are still the core of their solution. The novel pipelining technique can also be applied on our solution to hide source-to-source transformations. We have a weaker motivation because our transformations are not affected by different inputs like in the case of on-CPU TDR. Assuming CPU overheads are completely hidden by the pipeline, a rough comparison can be made by only looking at the GPU execution

time. However, for memory-intensive applications like GDT, the CPU-GPU pipeline will be hard to configure, and the old mapping may have to be used.

On-CPU TDR cannot be directly used on branches with compile-time unknown conditions, and multiple branches in the same kernel may require conflicting mappings. Online inspection can be utilized at the cost of kernel-splitting, which increases management complexity and memory traffic. As a result, on-CPU TDR often relies on heuristics such as sorting input data according to numerical values, which may sometimes introduce more divergence or break the logical thread groupings. To pinpoint and tackle true divergence, Liang et al. proposed to incorporate profiling runs and their performance model [15], which is not flexible with different inputs.

There are also hardware and software solutions that aim to increase divergence tolerance, rather than to eliminate source of the problem. Meng et al. [16] demonstrated how wavefront subdivision helps increase divergence tolerance, and a similar principle called *independent thread scheduling* has been incorporated into the NVIDIA Volta architecture. With this new hardware design, branch divergence does not disappear because still only one instruction is allowed at any time. In fact, our solution can benefit from the more flexible sub-workgroup synchronization, especially when our algorithms are applied on nested branches. Software solutions [6, 11] rely on compiler techniques to move shared instructions out of divergent paths. These approaches are compatible and complementary to ours.

6 CONCLUSION

In this article, we propose on-GPU TDR as a novel compiler-assisted runtime solution for fine-grained branch divergence reduction. Unlike traditional TDR solutions, our on-GPU TDR algorithms can be directly applied on branches whose conditions are unknown at compile-time. Each target branch in a kernel can be treated separately, changing the paradigm of TDR. Three stages of evaluation are conducted on two generations of GPUs from both NVIDIA and AMD, and some key findings are listed below.

- For the micro-benchmark, both *DGI* and *HoT* achieved theoretical speedups on all four GPUs. We calculated control flow efficiency from N980 runtime statistics, which turned out to be a good indicator of divergence severeness and potential speedup.
- For the seven third-party divergent benchmarks, WAGNERR averaged 71% of the potential performance improvement. As expected, overhead of *HoT* was smaller for two-path branches, while *DGI* is more suitable for many-path branches.
- We implemented GPGPU frontier benchmarks from three areas, each with a different memory access pattern. The WAGNERR-processed versions were faster than the on-CPU TDR versions in all cases.
- It was found that on-GPU TDR has an overall positive impact on GPU memory performance, reducing L2-cache misses by 8% on average.
- Comparing the GPU time of WAGNERR-processed kernels and the on-CPU TDR version, we estimated the time overhead of each round of on-GPU TDR to be less than 0.05ms.
- Development of GPU hardware would continue to reduce on-GPU TDR overhead. Especially, the independent thread scheduling in NVIDIA Volta architecture would allow more flexible application of on-GPU TDR.

In the future, we will implement a parser for CUDA or incorporate WAGNERR into auto-parallelizers. With the extra disclosed runtime statistics from CUDA and further study on hardware specifications, WAGNERR can be optimized in greater details, and possibly for different GPU models. Divergence identification will rely less on CFE profiling. We will also optimize the on-GPU

TDR algorithms and focus on the recursion scenario. We hope on-GPU TDR can facilitate more divergent applications on the GPU platform.

ACKNOWLEDGMENTS

The authors would like to thank the reviewers for their time and efforts.

REFERENCES

[1] Andy Adinets. 2014. CUDA Pro Tip: Optimized Filtering with Warp-Aggregated Atomics. Retrieved April 22, 2018 from https://devblogs.nvidia.com/cuda-pro-tip-optimized-filtering-warp-aggregated-atomics/.

[2] Yuri Ardila, Natsuki Kawai, Takashi Nakamura, and Yosuke Tamura. 2013. BEMAP: BEnchmark for auto-parallelizer. Retrieved August 29, 2018 from http://jglobal.jst.go.jp/en/public/201302228262237597.

[3] Sebastian Breß, Max Heimel, Norbert Siegmund, Ladjel Bellatreche, and Gunter Saake. 2014. GPU-accelerated database systems: Survey and open challenges. In *Transactions on Large-Scale Data-and Knowledge-Centered Systems XV*. Springer, 1–35.

[4] James Brookhouse, Fernando E. B. Otero, and Michael Kampouridis. 2014. Working with openCL to speed up a genetic programming financial forecasting algorithm: Initial results. In *Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation*. ACM, 1117–1124.

[5] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the IEEE International Symposium on Workload Characterization, 2009 (IISWC'09)*. IEEE, 44–54.

[6] Bruno Coutinho, Diogo Sampaio, Fernando Magno Quintao Pereira, and Wagner Meira Jr. 2011. Divergence analysis and optimizations. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT'11)*. IEEE, 320–329.

[7] Zheng Cui, Yun Liang, Kyle Rupnow, and Deming Chen. 2012. An accurate GPU performance model for effective control flow divergence optimization. In *Proceedings of the 2012 IEEE 26th International Parallel & Distributed Processing Symposium (IPDPS'12)*. IEEE, 83–94.

[8] Jack J. Dongarra, Hans W. Meuer, and Erich Strohmaier. 1994. Top500 supercomputer sites. Retrieved February 11, 2018 from https://www.top500.org/.

[9] Wilson W. L. Fung and Tor M. Aamodt. 2011. Thread block compaction for efficient SIMT control flow. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA'11)*. IEEE, 25–36.

[10] Wilson W. L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. 2007. Dynamic warp formation and scheduling for efficient GPU control flow. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 407–420.

[11] Tianyi David Han and Tarek S. Abdelrahman. 2011. Reducing branch divergence in GPU programs. In *Proceedings of the 4th Workshop on General Purpose Processing on Graphics Processing Units*. ACM, 3.

[12] Christoph Hartmann, Ralph Mader, Lothar Michel, Christos Ebert, and Ulrich Margull. 2017. Massive parallelization of real-world automotive real-time software by GPGPU. In *Proceedings of the 30th International Conference on Architecture of Computing Systems (ARCS'17)*. VDE, 1–8.

[13] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM International Conference on Multimedia*. ACM, 675–678.

[14] Konstantinos Krommydas, Wu-chun Feng, Christos D. Antonopoulos, and Nikolaos Bellas. 2016. Opendwarfs: Characterization of dwarf-based benchmarks on fixed and reconfigurable architectures. *Journal of Signal Processing Systems* 85, 3 (2016), 373–392.

[15] Yun Liang, Muhammad Teguh Satria, Kyle Rupnow, and Deming Chen. 2016. An accurate GPU performance model for effective control flow divergence optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35, 7 (2016), 1165–1178.

[16] Jiayuan Meng, David Tarjan, and Kevin Skadron. 2010. Dynamic warp subdivision for integrated branch and memory divergence tolerance. *ACM SIGARCH Computer Architecture News* 38, 3 (2010), 235–246.

[17] CUDA Nvidia. 2008. Programming guide. Retrieved August 29, 2018 from https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[18] Terence J. Parr and Russell W. Quong. 1995. ANTLR: A predicated-LL (k) parser generator. *Software: Practice and Experience* 25, 7 (1995), 789–810.

[19] Daniele Pianu, Roberto Nerino, Claudia Ferraris, and Antonio Chimienti. 2016. A novel approach to train random forests on GPU for computer vision applications using local features. *The International Journal of High Performance Computing Applications* 30, 3 (2016), 290–304.

[20] Timothy J. Purcell, Craig Donner, Mike Cammarano, Henrik Wann Jensen, and Pat Hanrahan. 2003. Photon mapping on programmable graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*. Eurographics Association, 41–50.

[21] Minsoo Rhu and Mattan Erez. 2012. CAPRI: Prediction of compaction-adequacy for handling control-divergence in GPGPU architectures. In *ACM SIGARCH Computer Architecture News*, Vol. 40. IEEE Computer Society, 61–71.

[22] Sangmin Seo, Gangwon Jo, and Jaejin Lee. 2011. Performance characterization of the NAS parallel benchmarks in OpenCL. In *Proceedings of the 2011 IEEE International Symposium on Workload Characterization (IISWC'11)*. IEEE, 137–148.

[23] John E. Stone, David Gohara, and Guochun Shi. 2010. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in Science & Engineering* 12, 1–3 (2010), 66–73.

[24] John A. Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W. Hwu. 2012. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing* 127 (2012).

[25] Weibin Sun and Robert Ricci. 2013. Fast and flexible: Parallel packet processing with GPUs and click. In *Proceedings of the 9th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. IEEE Press, 25–36.

[26] Aniruddha S. Vaidya, Anahita Shayesteh, Dong Hyuk Woo, Roy Saharoy, and Mani Azimi. 2013. SIMD divergence optimization through intra-warp compaction. *ACM SIGARCH Computer Architecture News* 41, 3 (2013), 368–379.

[27] Eddy Z. Zhang, Yunlian Jiang, Ziyu Guo, and Xipeng Shen. 2010. Streamlining gpu applications on the fly. In *Proceedings of the ACM International Conference on Supercomputing (ICS'10)*. 115–125.

[28] Eddy Z. Zhang, Yunlian Jiang, Ziyu Guo, Kai Tian, and Xipeng Shen. 2011. On-the-fly elimination of dynamic irregularities for GPU computing. In *ACM SIGARCH Computer Architecture News*, Vol. 39. ACM, 369–380.

Received February 2018; revised June 2018; accepted July 2018