Submitted to DCMP12 Special Issue

HARP: Harnessing Inactive Threads in Many-Core Processors

AHMAD LASHGAR, University of Tehran
AHMAD KHONSARI, University of Tehran
AMIRALI BANIASADI, University of Victoria

SIMT accelerators are equipped with thousands of computational resources. Conventional accelerators, however, fail to fully utilize available resources due to branch and memory divergences. This underutilization is manifested in two underlying inefficiencies: pipeline width underutilization and pipeline depth underutilization. Width underutilization occurs when SIMD execution units are not entirely utilized due to branch divergences. This affects lane activity and results in SIMD inefficiency. Depth underutilization takes place when the pipeline runs out of active threads and is forced to leave pipeline stages idle. This work addresses both inefficiencies by harnessing inactive threads available to the pipeline. We introduce Harnessing inActive thReads in many-core Processors (or simply HARP) to improve width and depth utilization in accelerators. We show how using inactive yet ready threads can enhance performance. Moreover, we investigate implementation details and study microarchitectural changes needed to build a HARP-enhanced accelerator. Furthermore, we evaluate HARP under a variety of microarchitectural design points. We measure the area overhead associated with HARP and compare to conventional alternatives. Under Fermi-like GPUs, we show that HARP provides 10% speedup on average (maximum of 1.6X) at the cost of 3.5% area overhead. Our analysis shows that HARP performs better under narrower SIMD and shorter pipelines.

Categories and Subject Descriptors: **C.1.2 [Processor Architectures]:** Multiple Data Stream Architectures (Multiprocessors)

General Terms: Design, Measurement, Performance

Additional Key Words and Phrases: Many-core, Accelerator, Multi-threading, Branch divergence, Memory divergence

1. INTRODUCTION

Many-core accelerators (e.g. GPUs) are built to attain high throughput for massively-parallel workloads. GPUs employ hundreds of processing units in several cores to run groups of parallel scalar threads (referred to as warps) in lockstep. Following this single-instruction, multiple-thread (SIMT) approach has resulted in significant and steady performance growth in GPUs.

Modern programmable general-purpose GPUs rely on this performance growth to carry data-parallel computations on a wide variety of devices ranging from embedded devices [Mijat 2012; Imagination Technologies 2012] to high-performance computing systems [Yang et al. 2011].

Applications with enough thread level parallelism (TLP) and an adequate share of independent threads benefit highly from GPU organizations where warps formed by multiple threads travel through the pipeline. Other applications may benefit less from GPUs as a result of either low TLP or thread control and data dependences, often referred to as branch and memory divergence.

In this study we view each SIMD processor as a structure with two dimensions; *pipeline width* and *pipeline depth*. Pipeline width represents the number of parallel lanes in SIMD. Pipeline depth represents the number of stages in the core pipeline. This two-dimensional structure makes executing multiple threads in parallel possible. Tens of threads are grouped into warps and processed in lockstep to utilize the wide SIMD of core (*pipeline width*). Tens of warps are interleaved to occupy the *pipeline depth* and hide the memory access latency. Peak throughput of GPUs is achieved if both pipeline width and pipeline depth are fully utilized.

Author's addresses: a.lashgar@ece.ut.ac.ir, amirali@ece.uvic.ca, ak@ipm.ir

Table 1. Comparing the multithreading trend in NVIDIA GPUs. MT depth denotes the number of threads per chip. GFLOPs correspond to peak single-precision performance of MAD instruction. [NVIDIA Corp. 2012c; NVIDIA Corp. 2012d]

	MT depth	GFLOPs	MTDepth/GFLOPs	Register/chip
GeForce GTX 285	30720	704.2	43.6	480K
GeForce GTX 580	24576	1581.1	15.5	512K
GeForce GTX 680	16384	3409.9	4.8	512K

Peak throughput is rarely achieved. Branch divergence reduces the number of active threads in warps, degrading width utilization. Memory divergence stalls an entire warp, degrading depth utilization.

Previous works have proposed solutions addressing the above inefficiencies. Some earlier studies have enhanced depth utilization through TLP improvement [Fung et al. 2007; Meng et al. 2010]. Other solutions have improved width utilization [Fung and Aamodt 2011; Narasiman et al. 2011; Rhu and Erez 2012]. The first group improves depth utilization while risking width utilization. The second group improves width utilization at the cost of TLP reduction, risking depth utilization.

The trend in NVIDIA GPUs indicates a reduction in chip multithreading-depth (number of concurrent threads) per GFLOPs over recent generations. As reported in Table 1, GTX 680 comes with significantly lower chip multithreading per GFLOPs (MTDepth/GFLOPs) compared to GTX 285. This can be due to the fact that deep multithreading harms locality at several levels including on-chip caches [Meng et al. 2010; Jog et al. 2013] and DRAM scheduling [Yuan et al 2009]. While aggressive multithreading fails to guarantee better performance, in order to utilize ever-growing computation capability of the chip, future GPU microarchitectures should exploit available TLP more effectively. To this end we propose HArnessing inactive thReads in many-core Processors (or HARP) as an alternative microarchitectural solution aiming at using the available TLP more efficiently and effectively.

HARP builds on the observation that, frequently enough, there are occasions where the GPU is suffering from low width and depth utilization while there are a high number of overlooked inactive and ready threads.

In summary we make following contributions:

We study conventional and previously suggested control-flow mechanisms and investigate their impact on width and depth utilization. By isolating the two dimensions we provide better understanding of current and future GPU challenges. This view effectively describes the source of underutilization under different mechanisms.

We categorize the status of concurrent threads during pipeline idle cycles, highlighting different opportunities for pipeline depth utilization enhancement. For example, we show that during pipeline idle cycles, a significant share of threads are waiting at inter-block synchronizers, diverging paths, or re-convergence points.

We propose HARP as a solution that exploits stalled TLP to enhance pipeline depth utilization and width utilization. We propose an efficient microarchitecture for HARP and evaluate the associated design space to study different performance/overhead tradeoffs.

We evaluate performance under HARP and compare it to previous studies. We analyze the sensitivity of our findings under various microarchitectural changes. We evaluate the area overhead of HARP.

The rest of this paper is organized as follows. Section 2 reviews background. Section 3 presents our motivation. Section 4 introduces HARP. In Section 5 we review related works. Section 6 presents methodology. In Section 7 we evaluate HARP and analyze the results. In Section 8 we evaluate HARP's overhead. Finally, in Section 9 we offer concluding remarks.

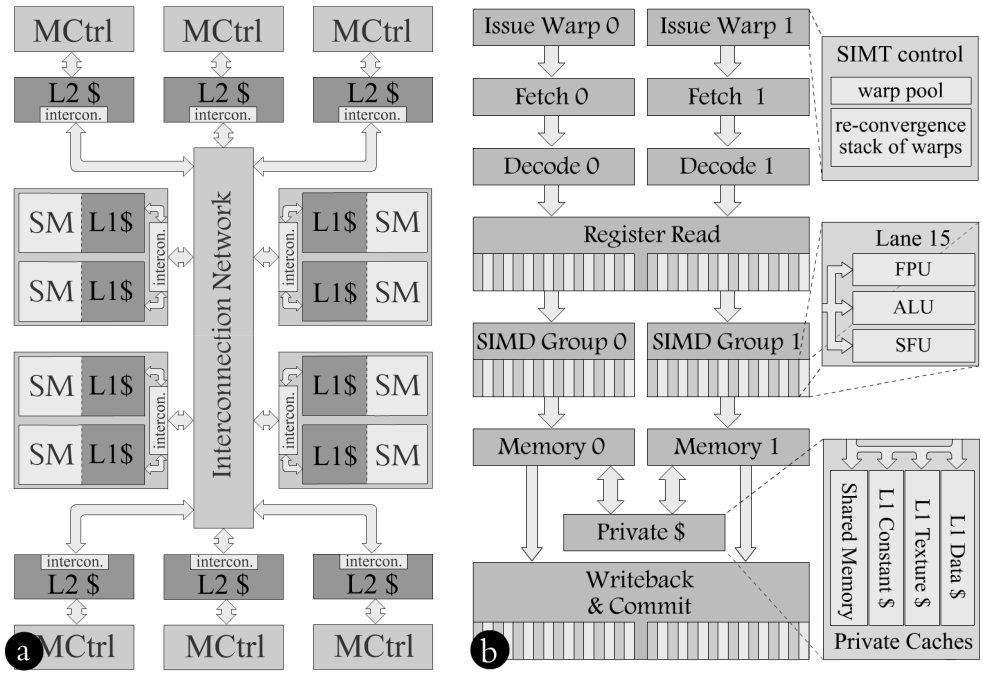

Figure 1. (a) The many-core accelerator used in this study. (b) Stream Multiprocessor (SM) microarchitecture. We assume 16 lanes per SIMD group.

2. BACKGROUND

We assume a GPU-like accelerator executing CUDA/OpenCL workloads [NVIDIA Corp. 2012a; Khronos Group 2012] similar to NVIDIA Fermi [Wittenbrink et al. 2011]. Figure 1(a) shows the top-level design of this accelerator. Upon an L1 cache miss, a request is sent to the last-level L2 cache. The L2 cache is partitioned into six banks; one bank per memory controller. If the L2 cache misses the data, the request is sent to the corresponding memory controller to access the off-chip DRAM. Streaming Multiprocessors (SMs) are SIMD multithreaded processing cores. To hide the associated latency, SM interleaves the execution of hundreds of threads assigned to it in the thread-block.

Figure 1(b) depicts the internal microarchitecture of SM. SM pipeline frontend employs two parallel independent single-issue warp schedulers. Each scheduler interleaves the execution of 768 threads and issues instructions on its dedicated SIMD group. Each SIMD group is a 16-lane SIMD engine executing the same instruction for 16 different threads in parallel. SIMD groups are in-order and fully-pipelined. Each SM keeps the context, including registers and shared memory, for the total of 1536 concurrent threads. 32 threads are grouped into one *warp* and execute over two cycles. Every two cycles, issue warp stage of each scheduler selects and issues a ready warp from the warp pool if the pipeline is not stalled. Register Read stages read operands of active threads in one-half of warp concurrently by accessing a highly banked register file. One 4-port (three reads/one write) register file is virtually associated with each lane to conserve register file bandwidth demand. Each thread in the warp has a unique lane identifier (ID), which does not change over thread lifetime. Therefore all registers of a thread are allocated in the register file associated with the thread's lane.

Grouping threads in a warp enhances SIMD efficiency as threads execute with the same pace. Moreover, this makes it possible to use a simplified scheduler (e.g., round-robin), which runs at warp-granularity. On the other hand, warp-level granularity

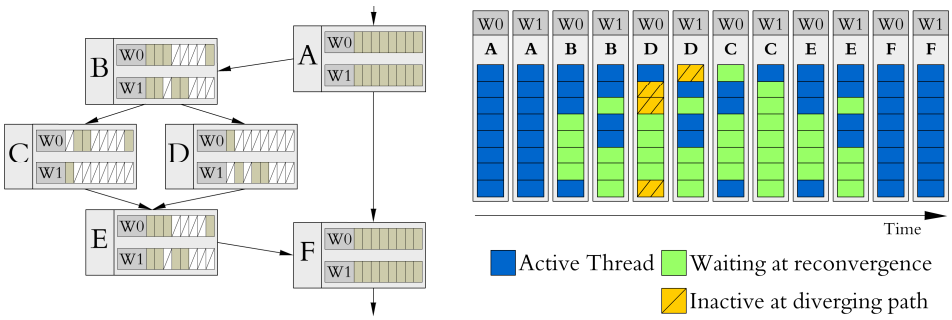

Figure 2. Interleaved execution of two warps (W0 and W1) under SBR. Negative consequences: Reducing the number of active threads in diverging paths leaving some lanes idle. If active threads (at basic block B, C, D or E) execute long latency instructions, there will be no ready warp to hide this latency. Opportunities: i) merging W0 and W1 upon executing C to the reduce number of idle lanes, and ii) exploiting inactive/waiting threads to hide the latency if all warps are pending.

imposes two types of divergences, which can potentially harm performance; memory divergence and branch divergence. *Memory divergence* may occur upon executing a memory instruction. Upon accessing the cache, a warp may diverge into hit-threads and miss-threads. Hit-threads are threads that find the data in cache. However, lock-step warping execution stalls force hit-threads to wait for miss-threads, which have missed the data in cache. *Branch divergence* may occur upon executing conditional branch instructions. If branches in a warp behave differently, the warp diverges into taken and not-taken threads. Control-flow mechanism is needed to handle this type of divergence. Current GPUs from NVIDIA use stack-based re-convergence (SBR) to manage branch divergence [Collange 2011]. SBR effectively executes diverging paths serially and joins the diverged threads at the end of the diverging paths.

In the remainder of background we present an overview of the previously proposed control-flow mechanisms. In the next section, we discuss how each mechanism impacts pipeline depth/width utilization.

2.1 Stack-based Re-convergence

Upon executing a branch instruction, threads of a warp can have different outcomes. As a result, threads can diverge to different program counters (PCs). Current accelerators use stack-based re-convergence (SBR) [Fung et al. 2007] to store program counters associated with the diverged threads as logical program counters of the warp.

SBR executes the diverged threads serially. SBR uses a stack per warp to facilitate this. Each entry of stack has three fields: logical program counter (PC), corresponding threads, and re-convergence point program counter (RPC). The re-convergence point is where all diverging paths eventually rejoin.

Upon encountering a diverging branch, the PC of the top entry is replaced with the re-convergence point of the branch. Based on the number of diverging sides, one or two entries are pushed to keep track of the diverging paths. For these new entries, RPC is stored to prevent execution of the diverged threads beyond the re-convergence point. The PC of the top entry is the physical program counter of the warp. As the PC of the top entry reaches its RPC, the top entry is popped and execution continues following the new top of the stack. A detailed operation example of SBR can be found in [Fung et al. 2007]. Figure 2 depicts the interleaved execution of two warps inside diverging paths of the given control-flow graph.

2.2 Dynamic Warp Formation

Dynamic Warp Formation (DWF) [Fung et al. 2007] terminates warps upon commit; regrouping their threads into new/existing warps to maximize warp occupation. DWF performs similar to SBR for branches behaving coherently across all threads of the warp. Upon branch divergence, threads are grouped into different warps to interleave the execution of diverging paths. DWF occupies the empty lanes of the diverged warps by placing the other diverged threads into empty lanes. DWF employs a small lookup table to search for merging opportunities.

The main challenge with DWF is that the diverged threads may never re-converge again. Therefore, DWF may result in constructing multiple combinable less-occupied warps running at different paces. DWF can jeopardize pipeline width utilization due to re-convergence misses.

2.3 Large Warp Microarchitecture

Large Warp Microarchitecture (LWM) [Narasiman et al. 2011] groups hundreds of threads into larger warps. LWM keeps track of divergence using a larger re-convergence stack. Unlike SBR, LWM compacts active threads at the top of stack into multiple sub-warps (warps as wide as SIMD), maximizing warps' occupations. Sub-warps are constructed at the issue warp stage. Sub-warps are executed and synchronized at each instruction. This policy effectively improves the number of active lanes in the SIMD group. LWM, however, can result in stalling a high number of threads due to large warp synchronization, which in turn exacerbates the negative consequences of branch/memory divergence.

3. MOTIVATION

GPUs achieve high throughput by utilizing the SM's SIMD pipeline. Achieving steady and high throughput computation requires effective utilization of both pipeline depth and pipeline width (SIMD width).

Threads belonging to the same warp utilize the pipeline width (also referred to as lane activity). Concurrent warps running on different SM pipeline stages occupy the pipeline depth. In this section, we first show how SBR impacts pipeline depth and width utilization. Second, we show that there is a large group of stalled threads, which can be activated to improve pipeline depth utilization. We investigate this opportunity and comment on how it can be used to achieve high pipeline depth and width utilization simultaneously.

3.1 Pipeline Depth Utilization

SIMD stays idle if the pipeline front-end does not find any eligible instruction to issue. This can occur frequently as a result of instruction dependency on long latency load instructions. We measure the pipeline depth utilization according to following equation:

$$\text{pipelineDepthUtilization} = 1 - \text{idleCycles} \qquad (1)$$

Where the *idleCycles* is the rate of cycles where SIMD has no active lane. When warp-level parallelism is not high enough to hide memory latency, depth utilization drops.

Figure 3 reports pipeline depth utilization for different benchmarks under SBR (see Section 6 for methodology). Depth utilization is high in benchmarks having enough active warps to hide memory latency (CP, HSPT, and PF). However, in many benchmarks, the available warp-level parallelism is not enough to hide memory latency effectively. Accordingly, the pipeline experiences idle cycles frequently, degrading depth utilization.

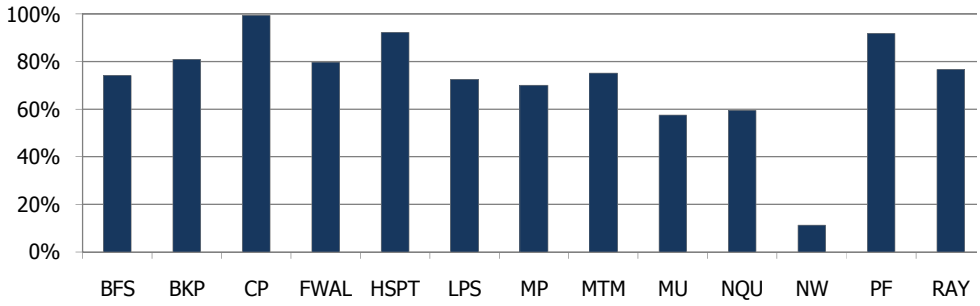

Figure 3. Pipeline depth utilization.

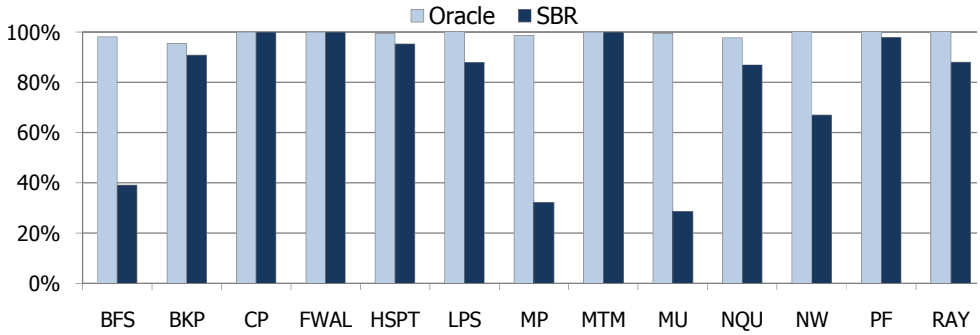

Figure 4. Pipeline width utilization.

3.2 Pipeline Width Utilization

Pipeline width utilization drops due to 1) branch divergence within a warp or 2) inherently, having few threads in a warp. The former happens frequently in applications with data-dependent control-flow divergence. The latter happens when the thread-block size is less than SIMD width. As a result, warps end up having a few active threads, underutilizing the pipeline width.

We use lane activity (below) to measure pipeline width utilization:

$$\text{lane activity} = 100 \times \frac{\sum_{i=1}^{\text{n\_issues}} \text{act\_thds[i]}}{\text{n\_issues} \times \text{SIMDWidth}} \qquad (2)$$

where *n_issues* is the number of cycles where at least one SIMD lane is active, *act_thds[i]* is the number of active lanes in the *i*th cycle, and *SIMDWidth* is the number of lanes in SIMD. Notice that the equation (2) ignores idle cycles focusing solely on width utilization.

SBR-based solutions lose lane activity as they mask diverged threads not belonging to the current physical PC. In Figure 4 we report lane activity for the benchmark set used in this study. The left bar for each benchmark (Oracle) represents a perfect control-flow solution. To model the Oracle mechanism, we perform two simulation steps for each kernel. First we obtain a trace file of the individual PCs that each thread executes. Then, we group the same PCs of different threads to form warps with maximum lane activity. The grouping also operates under two constraints: i) threads cannot change lane in warp, and ii) only threads within the same thread-block can be grouped into the same warp. This unrealistic approach provides an upper bound for lane activity under dynamic regrouping methods. As reported in Figure 4, in benchmarks with coherent control-flow, pipeline

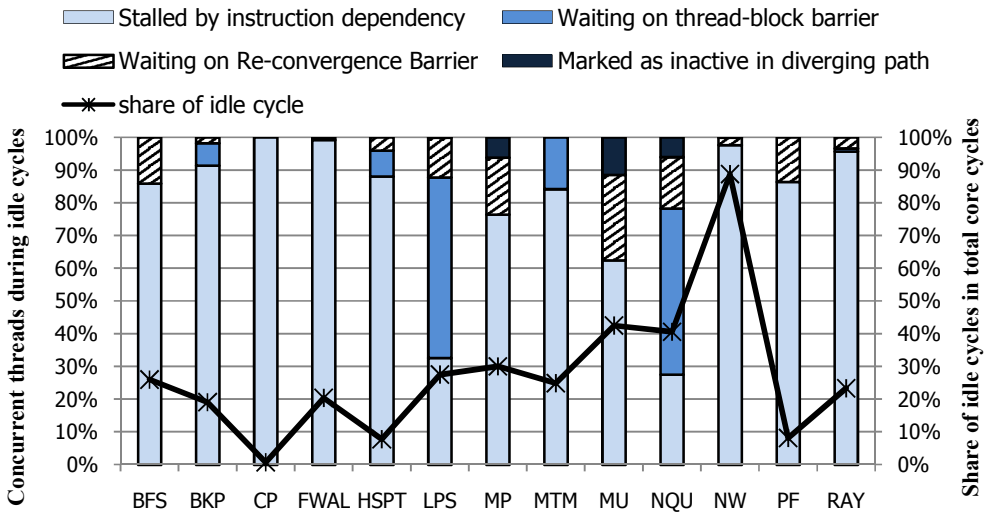

Figure 5. Concurrent threads breakdown during SM idle cycles.

width is fully utilized (e.g. CP and MTM). However, data-dependent control-flow divergence frequently occurs in many benchmarks resulting in a significant drop in width utilization (e.g. BFS, MU, and MP).

3.3 Unexploited TLP

GPUs lose performance when SMs become idle. Figure 5 presents the share of SM idle cycles in the overall program execution time. As reported, up to 89% of the cycles can be idle. Figure 5 also reports the status of concurrent threads during idle cycles. During idle cycles, the majority of threads are stalled due to instruction dependency, usually a long latency memory fetch. A considerable number of threads wait at intra-thread-block barriers for other threads of thread-block. The remaining threads are stalled due to SBR's branch divergence management policy; *i) threads marked as inactive in diverging paths* or *ii) threads waiting at re-convergence barrier*. These threads can account for as much as 37% of the entire thread pool (in MU benchmark). Below we discuss these groups in more detail and highlight the opportunity available to improve pipeline depth utilization.

3.3.1 Stalled threads

Inactive threads. These threads are stalled due to the re-convergence stack organization (not instruction or control dependency). The stack only activates one side of the divergence leaving the other sides' threads inactive. The number of inactive threads can grow rapidly with warp size and/or the number of nested two-sided conditional branches. This group of threads may include many ready, yet inactive threads.

Re-converging threads. SBR's re-convergence policy requires keeping threads of the same warp waiting at the re-convergence point. This waiting period is longer when the diverging paths of a branch are imbalanced or there is a costly unstructured control-flow [Diamos et al. 2011].

3.3.2 Opportunity

An SIMD group may be idle despite having a significant number of ready but stalled threads. Inactive threads or waiting at re-convergence threads, if activated, can proceed to improve *pipeline depth* utilization. Below we argue how each type of threads can be exploited.

First, inactive threads can be grouped into new warps to execute diverging paths independently in parallel. This could enhance the pipeline depth utilization by including the ignored ready threads in the ready warp pool. It should be noted that executing inactive threads beyond the re-convergence point can compromise lane activity (*width utilization*). This may occur as a result of multiple combinable warps executing the same re-convergence point. Therefore, enhancing pipeline depth utilization has to stay aware of re-convergence benefits to assure maintaining pipeline width utilization.

Second, the threads waiting at re-convergence point can form new warps to proceed and improve depth utilization. In order to enhance depth utilization while protecting lane activity, instead of relaxing all re-convergence synchronization, we rely on synchronizing fewer threads (as many as SIMD width). Consequently, we reduce the number of threads waiting at re-convergence point while preventing lane activity loss.

4. HARP

We propose HARP to improve pipeline depth and width utilization. HARP takes the following measures to achieve this goal:

Pipeline width: HARP regroups diverged threads into new warps. The newly dynamically constructed warps use threads from different warps to increase lane activity. At the same time, HARP uses re-convergence points to protect against low SIMD efficiency.

Pipeline depth: HARP improves TLP to increase depth utilization. All diverged threads are activated so threads from diverged paths can execute simultaneously. Moreover, threads are grouped into small warps (as wide as SIMD width) to reduce the number of threads waiting at re-convergence point.

Below we describe HARP in more detail. In Section 4.1, we provide a high-level description. In Section 4.2, we present HARP's microarchitecture details. In Section 4.3, we present an operation example.

4.1 High-Level Description

HARP exploits existing stalled threads to improve TLP. HARP constructs up to two independent schedulable warps after a branch divergence. Under HARP, both divergence sides remain active and can potentially hide the other side's latency, improving depth utilization. The diverged warps of the same branch are synchronized at the re-convergence point to revive pre-divergence warps. HARP constructs smaller warps (as wide as SIMD width) to mitigate the number of threads stalling at the re-convergence point.

HARP maintains lane activity (in the presence of branch divergence) by dynamically regrouping diverged threads from different warps into new warps. Accordingly, HARP retains recently constructed warps in a small lookup table, separated from the scheduling pool. Upon warp divergence and before constructing new warps, HARP searches the lookup table for merging possibilities. Diverged threads are merged with warps with the same program counter and without lane conflict. Otherwise, a maximum of two new warps are constructed using the diverged threads failing to merge with existing warps in the lookup table.

Re-convergence is necessary to protect lane activity. HARP employs re-convergence by taking the following steps. Upon branch divergence, HARP

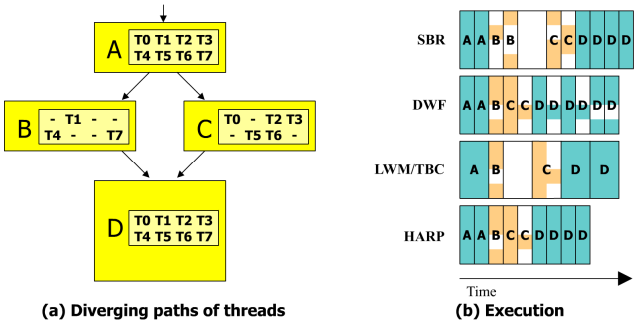

Figure 6. Tentative comparison of HARP execution to previous works. Adapted from a figure by Fung and Aamodt [2011]. Code block D has two instructions whereas other code blocks have a single instruction. We assume that each instruction takes one time unit to execute a single thread. We also assume block B executes a long latency instruction (e.g., memory access) taking two time units. SBR suffers from low lane activity under diverging paths B and C. Moreover, pipeline stays idle during the long latency access in B. DWF improves lane activity by regrouping diverged threads into new warps and hides B's long latency access using threads from C. However, DWF misses the re-convergence opportunity and executes multiple combinable warps at D. LWM keeps a large number of threads at the same pace and compacts stack's top active mask to improve width utilization. However, LWM still suffers from depth underutilization when running B. HARP dynamically regroups diverged threads (improving width) and interleaves the execution of diverging paths (improving depth utilization). HARP achieves higher pipeline depth utilization by reducing the inter-threads synchronization overhead. However, HARP's width utilization can be lower than LWM if dynamic regrouping finds less merging opportunities compared to compaction.

invalidates the diverging warp (also referred to as the pre-divergence-warp here) and constructs two new warps at diverging paths. Concurrently, a barrier is reserved to rejoin these warps at the re-convergence point. The barrier is unique for the re-convergence point and diverged threads. Subsequently, the scheduler interleaves the execution of these new warps, executing different diverging paths independently. Once new warps reach the re-convergence point, the scheduler invalidates them. At this point, the barrier associated with the warp and re-convergence point marks the threads of warps reaching the barrier as 'reached'. When all threads of the pre-divergence-warp have reached the re-convergence point, the barrier marks all threads as 'reached'. At this time, the barrier reincarnates the pre-divergence-warp and the scheduler continues the execution of the warp at the re-convergence point.

Figure 6 provides better understanding of how pipeline depth/width utilization impacts performance. Diverging paths (including threads) are presented in 6(a). The execution under different control-flow mechanisms is presented in 6(b). We compare SBR, DWF, LWM, TBC, and HARP. We assume eight threads per warp for LWM and TBC (the other mechanisms have four threads per warp). LWM and TBC appear to behave similarly since each diverging path has a single instruction.

4.2 Microarchitecture

Figure 7 shows HARP implementation for a single SM. The implementation consists of 4 tables: First-Level warps (❶), Second-Level warps (❷), Lookup tables (❸), and Re-convergence Barriers (❹). In order to achieve low-overhead scheduling, HARP stores the active warps in two tables (❺) (i.e. First-Level and Second-Level). First-Level (❶) stores a limited number of ready warps scheduled for execution. HARP stores the remaining portion of active warps in Second-Level (❷). This group of warps eventually moves to First-Level table for execution. Lookup tables (❸) are used to

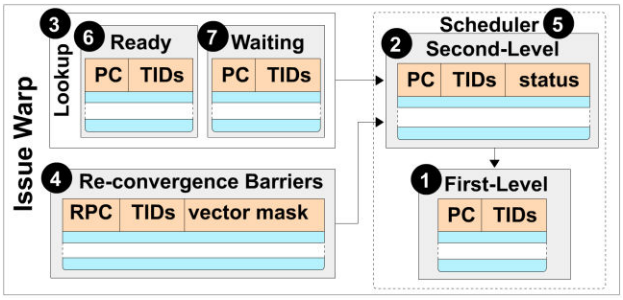

Figure 7. Issue warp stage of HARP control-flow mechanism.

store diverged threads waiting to be merged. Warps in Lookup tables depart to Second-Level eventually. Re-convergence Barriers table (❹) reserves a barrier to synchronize the threads of diverged warps at the re-convergence point. Each entry of the table is a barrier and consists of re-convergence program counter (RPC), identifiers of the threads that must be synchronized at the RPC, and the vector mask indicating which threads have reached the barrier. Below we provide further information on Lookup and Re-convergence Barriers tables. Then we discuss table-to-table transition policies and HARP's design space.

4.2.1 Lookup

The Ready-Lookup table (❻) maintains the recently diverged warps, which are intended to occupy the empty lanes of upcoming diverged warps if possible[1]. Merging diverged threads to any warp in the Ready-Lookup table is subject to: 1) the merging warp should have the same program counter as the diverged thread, and 2) the merging warp should not have lane conflict with the diverged thread.

Merging opportunities increase if the warps remain in Ready-Lookup table longer. On the negative side, keeping warps for long period deprives the pipeline from some ready warps, lowering depth utilization. In order to maintain merging opportunities under such circumstances we used Waiting-Lookup table (❼) and take advantage of long latency memory accesses. Warps residing in First-Level depart to Waiting-Lookup table upon decoding a global memory access instruction. Due to the low average hit rate of the L1 and L2 caches (respectively 69% and 57% for workloads evaluated in this study), the instruction completion usually can take hundreds of cycles during which the warp remains in Waiting-Lookup table. Other warps which execute the same instruction can merge with this warp (under the conditions used for Ready-Lookup table). Waiting-Lookup table effectively improves lane activity where diverged warps execute memory instructions. The warps in Waiting-Lookup table depart back to Second-Level eventually, when the pending memory access completes.

4.2.2 Re-convergence Barriers

Each entry in the re-convergence table is a barrier synchronizing a group of threads at the RPC. The barrier maintains a vector mask tracking the threads that have reached the RPC. Once all threads reach the RPC, the reincarnated warp is send to Second-Level scheduler, turning in to an active warp.

[1] Merging diverged threads can result in constructing new warps from arbitrary threads. Therefore, implicit synchronization is not guaranteed for all threads.

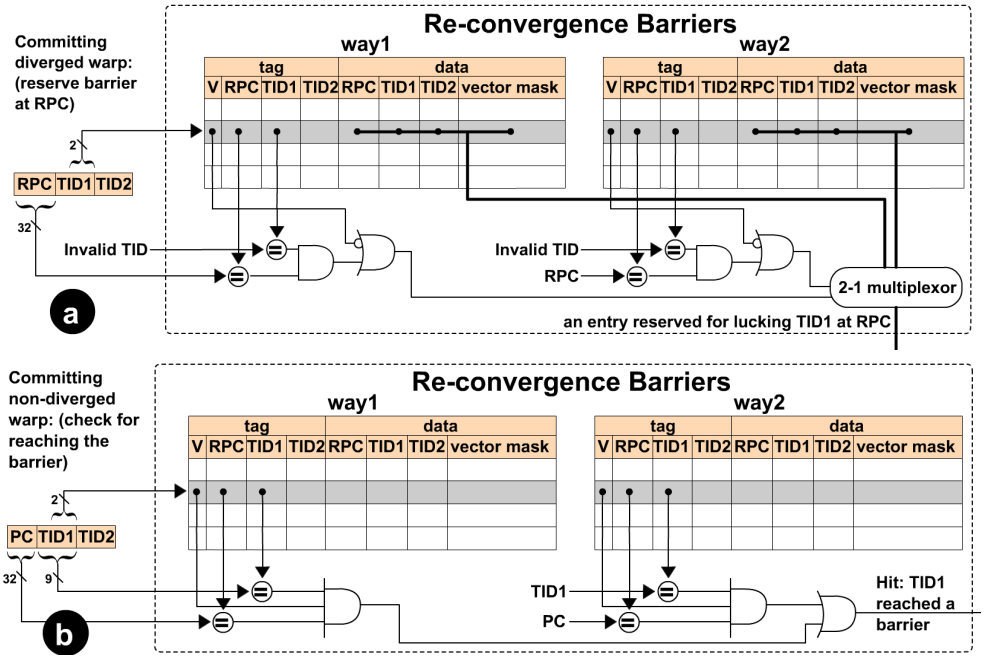

Figure 8. The architecture of an 8-entry two-way set-associative Re-convergence Barriers table for 2-wide SIMD. We assume 1024 concurrent threads so each TID is 9-bit. (a) Operations upon committing diverged warp. (b) Operations upon committing non-diverged warp.

HARP detects threads reaching the barrier upon commit. Threads belonging to the committing warp search their common next PC in Re-convergence Barriers table. If no match is found, the next PC is not the re-convergence point for the committing threads (the committing warp's status is updated as ready in Second-Level so the warp can proceed further). Otherwise, threads have reached the re-convergence barrier. The reaching threads are marked in the matching barriers and the committing warp is invalidated (similar to the pop operation in re-convergence stack under SBR).

Figure 8 presents Re-convergence Barriers. Each thread maps to a set in the table according to the least-significant bits of TID (thread lanes do not change during thread's lifetime and are therefore excluded from TID). 8 (a) shows the operations upon committing the diverged warp. In this case, a barrier is reserved on RPC for each lane of the warp. If the lookup finds a valid matching entry, the TID of associated lane is modified. If the lookup reserves a new entry, both TID and RPC are modified. In case there is no matching/free entry, the thread fails to reserve the barrier at RPC and will miss the re-convergence chance at RPC. 8 (b) shows operations upon committing the non-diverged warp. The table is searched for barriers stalling the committing threads at PC. If the lookup finds a match, the thread is marked at the 'vector mask' as reached and is excluded from the committing warp. This thread waits on the barrier until other threads reach the barrier and are similarly marked at the vector mask. The figure presents operations for one thread. Performing the operation for multiple threads will require replicating most of the logic presented here.

In the architecture presented in Figure 8, threads of the committing warp may need to access different sets of the table. Accordingly, the table should have access ports as many as the warp size. An alternative to this multi-port single-bank design,

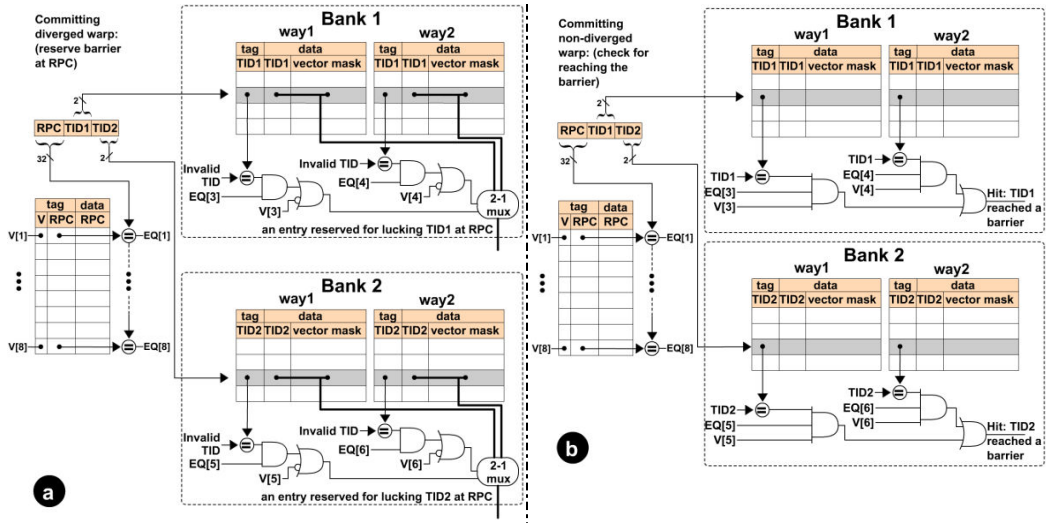

Figure 9. Under the same assumption as Figure 8, the multi-bank implementation of an 8-entry two-way set-associative Re-convergence Barriers table for 2-wide SIMD. (a) Operations upon committing diverged warp. (b) Operations upon committing non-diverged warp.

is implementing the table using multiple single-port banks. In Figure 9 we present this multi-bank design. Here the number of banks matches warp size. Since each thread accesses a dedicated bank, access conflicts are avoided. Each bank is associated with one lane of the warp and has a single port serving the corresponding warp lane. In Section 7.3.2 we report the performance impact of the associated timing overhead.

4.2.3 Transition policies

Warps depart from Lookup tables to Second-Level (lookup-to-second), from Re-convergence Barriers table to Second-Level (barrier-to-second), and from Second-Level to First-Level (second-to-first). Lookup-to-second transition should be performed just-in-time; early departure reduces the merging effectiveness and late departure reduces active warps (increasing the likelihood of an idle pipeline depth). We have evaluated complicated policies to achieve the right timing (e.g. keeping the warp in Lookup table so long there is a possibility to have a diverging warp at the same diverging path). However, we found that the best lookup-to-second transition policy is to transit once the Second-Level has a few ready warps (here less than 2). We use the FIFO policy to select the transiting warp. For barrier-to-second, there is no benefit to postpone the transition. Therefore, we perform the transition as soon as possible after all threads reach their barrier. For second-to-first, we use FIFO policy among the ready warps.

4.2.4 Design space

Under HARP, the size of warps and barriers are equal to SIMD width. The sizes of tables can vary, resulting in different costs and benefits. Our study shows that Lookup tables and Re-convergence Barriers table have the highest impact on performance and implementation cost. Therefore below we focus on these two tables.

Lookup tables are fully-associative and our evaluation shows that increasing the tables' sizes beyond 16-entry does not come with performance returns. We assume an 8-entry Ready-Lookup, and an 8-entry Waiting-Lookup table. If the diverging warp finds Lookup table full, the warp is bypassed to Second-Level and misses the chance of merging. Our evaluation shows that reducing Ready-Lookup size from 8 to 4

entries can degrade performance by 2%. Similarly, reducing Waiting-Lookup size from 8 to 4 entries can degrade performance by 6%.

Re-convergence Barriers table is a set-associative table. Each set is assigned to a range of threads (as discussed in Section 4.2.2, based on TID's least-significant bits). This design reduces the cost associated with search operations significantly while providing enough barriers compared to a fully-associative table. If the diverging warp finds the set full, no barrier is created and the diverged threads bypass the corresponding re-convergence point, possibly resulting in lane activity loss. Larger Re-convergence Barriers tables prevent this serialization. Our evaluation shows that employing a fully-associative Re-convergence Barriers table with more than 154 entries does not enhance performance. We assume a 128-entry 16-way set-associative table.

4.3 Operation Example

Figure 10 depicts an example demonstrating HARP operations. 10(a) represents a typical GPU kernel code exhibiting branch divergence. Code block A terminates with a diverging branch and code block D is the corresponding re-convergence point. Code block B and C are diverging paths of A, each executing a global memory instruction. 10(b) shows the diverging paths of 8 concurrent threads in the control-flow graph corresponding to the 10(a) kernel.

10(c) to 10(l) present how HARP operates during execution of the given code. To simplify the descriptions, we assume a unified scheduler, where First-Level and Second-Level warps merge in the same table (Scheduler). Two concurrent ready warps are ready to execute the last instruction of code block A (❶). The scheduler selects the next warp (W0) to issue and sends it to the pipeline. The next instruction of W0, which is a conditional branch at the end of A, is fetched in the next stage (❷). Similarly, in the subsequent cycle, the scheduler selects W1 as the next warp (❸). Branch divergence occurs upon committing W0 (❹). HARP performs three operations upon committing the diverged warp (W0): I) terminating W0 by invalidating W0's entry in Scheduler, II) reserving a barrier to re-converge the threads of W0 later at D (❺), and III) regrouping the diverged threads of W0 into new warps. For the barrier, the vector mask is set to all zero indicating none of the threads have reached the barrier. For regrouping, HARP searches Lookup table to merge the diverged threads with existing warps. Since the Lookup table is empty there are no merging opportunities and therefore two new warps are formed using the diverged threads (❻). Warps at the Lookup table eventually move to the scheduler table one by one every time the scheduler starts to run out of warps. In subsequent cycles, one of the warps in the Lookup table departs to the scheduler (❿). The departed warp is named W2 in the scheduler (❿). Similar to W0, W1 diverges upon commit. HARP performs three operations upon committing W1 (❼): I) invalidating W1, II) reserving a barrier to re-converge D (❽), and III) regrouping the diverged threads of W1. For the barrier, the vector mask is set to all zero. For regrouping, T4, T5, and T6 are regrouped into a new warp (❾) and T7 is merged into the empty lanes of the existing warp at code block C (❾). In the following cycle, the scheduler selects the only available warp (W2) for issue (⓬). W3, which has been residing at Lookup table, departs to the scheduler according to FIFO policy (⓫).

The remainder of the example demonstrates how re-convergence barriers synchronize threads. Upon committing W2 (⓯), Re-convergence Barriers table indicates T1 and T3 should wait for T0 and T2 on the barrier at D. Accordingly, HARP invalidates W2 and marks T1 and T3 as '1' in the vector mask of the barrier showing that T1 and T3 have reached D (⓰). The scheduler selects W3 as the next warp to issue (⓮). The only warp remaining at Lookup table moves to the scheduler

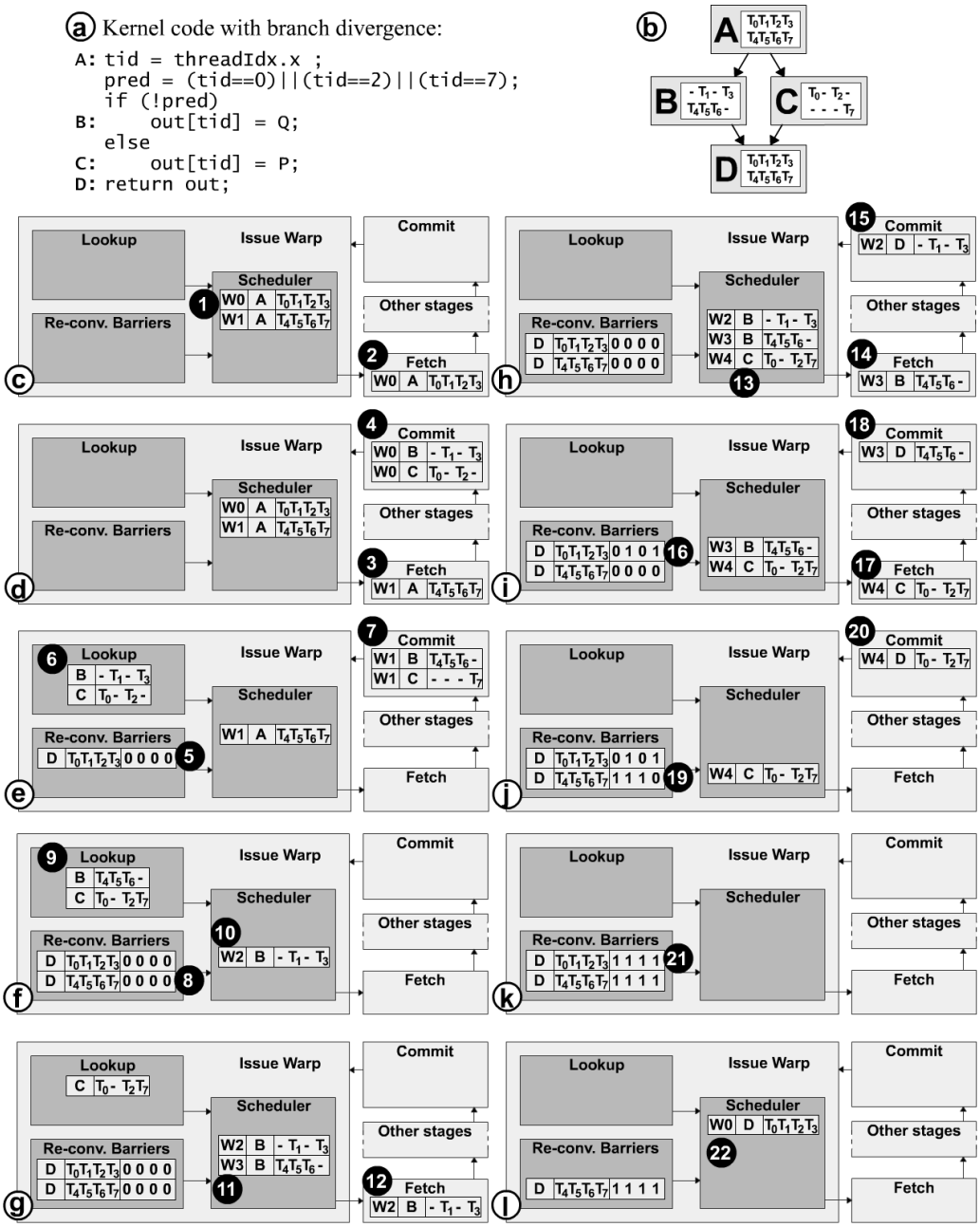

Figure 10. Operation example of HARP.

(labeled as W4) (⓭). Upon committing W3 (⓲), Re-convergence Barriers table indicates that T4, T5, and T6 should wait for T7 on the barrier at D. Accordingly, HARP invalidates W3 and marks T4, T5, and T6 as '1' in the vector mask of the barrier showing that they have reached D (⓳).A similar process continues with fetching W4 (⓱) and committing W4 at instruction D (⓴).

5. RELATED WORKS

Table 2 summarizes how previously suggested studies impact pipeline depth and width utilization. Previous studies have proposed three classes of solutions for

Table 2. A summary of previous studies' impact on pipeline depth/width utilization: '+' indicates addressing the associated parameter while '-' shows a possible negative impact on the issue compared to SBR.

	Pipeline Depth Utilization (Stalled threads)		Pipeline Width Utilization (lane activity)
	re-convergence waiting	inactive	
DWF	+no re-convergence	+no serialization	+dynamic regrouping -no re-convergence
DWS	+Heuristic	+Heuristic	-over subdivision
TBC	-Large warps	-Large warps	+Compaction
LWM	-Large warps	-Large warps	+Compaction
CAPRI	+Compaction-adequacy prediction -Large warps	-Large warps	+Compaction
SBI+SWI	+Co-issuing two diverging paths	+Co-issuing two diverging paths	+Co-issuing two diverging paths
DPE	+Activating two diverging paths	+Activating two diverging paths	NO_IMPACT
HARP	+Small warps	+no serialization	+dynamic regrouping

improving TLP (pipeline depth utilization): 1) relaxing all re-convergence points [Fung et al. 2007], 2) interleaving the execution of diverging paths to prevent serialization [Meng et al. 2010; Rhu and Erez 2013], and 3) selectively passing the threads waiting at re-convergence points [Meng et al. 2010; Rhu and Erez 2012]. In addition, there are two classes of solutions addressing lane activity (pipeline width utilization): 1) re-converging the threads belonging to the same warp earlier than the immediate post-dominator [Diamos et al. 2011; Fung and Aamodt 2011], or 2) using threads from other warps to replace the inactive lanes, forming new warps either through compaction [Fung and Aamodt 2011; Narasiman et al. 2011; Rhu and Erez 2012] or dynamic regrouping [Fung et al. 2007]. Below we briefly review important studies belonging to these classes.

Fung et al. [2007] proposed Dynamic Warp Formation (DWF) to address SM underutilization upon branch divergence. DWF targets lane activity using dynamic regrouping to keep the diverged threads of different warps in the same warp. Meng et al. [2010] proposed Dynamic Warp Subdivision (DWS) to improve memory level parallelism (MLP) and latency hiding without running more concurrent threads per SM. DWS attacks diverging path serialization and uses the inactive diverged threads to hide the latency of instructions issued from the active threads [Meng et al. 2010]. DWS proposes three heuristics (referred to as subdivision schemes) to achieve this. Thread Block Compaction (TBC) [Fung and Aamodt 2011] uses a stack per thread-block (instead of per warp). Lane activity improves as TBC compacts active threads at the top of the stack into multiple independent warps and synchronizes these warps at RPC (upon popping the top of re-convergence stack). Narasiman et al. [2011] proposed Large-Warp Microarchitecture (LWM) to address lane activity. LWM statically groups threads in large warps and executes warps in multiple cycles. As a negative consequence, some active threads diminish and some cycles are wasted in diverging paths. In order to prevent this, LWM compacts active threads at the top of the stack into warps as wide as SIMD width, and issues them one per cycle. Diamos et al. [2011] proposed Thread Frontier to re-converge at a re-convergence point earlier than the immediate post-dominator. This method is based on the fact that the threads of the same warp may execute the same diverging path multiple times in an unstructured control flow [Wu et al. 2012]. Rhu and Erez [2012] introduced CAPRI

on top of TBC to prevent unnecessary synchronizations at the end of basic blocks. CAPRI relies on a compaction-adequacy predictor to identify beneficial synchronizations and bypass others. CAPRI attacks lane activity and re-convergence waiting. Brunie et al. [2012] target lane activity by introducing instruction interleaving mechanisms (referred to as SBI, SWI, and SBI+SWI) to issue secondary instructions into the inactive lanes of primary issued warp. The secondary instructions can be co-issued from i) another warp, and/or ii) other diverging paths of the same warp. Each SIMD lane is configured dynamically to execute one of the instructions according to the operation mask. Rhu and Erez [2012] introduced Dual-Path Execution (DPE) model to interleave the execution of at most two parallel diverging paths per warp. DPE is implemented through simple modifications to the SBR re-convergence stack.

HARP is different from above studies as it targets lane activity improvement (pipeline width utilization) and TLP improvement (pipeline depth utilization) at the same time. Attacking each of these issues alone can affect the other. For example, DWF eliminates re-convergence waiting to enhance TLP through inactive and waiting threads. However, DWF can face situations where warps never re-converge resulting in consequences such as starvation eddies [Fung and Aamodt 2011]. DWF performs best under computational-intensive applications suffering from branch divergence. DWS eliminates serialization and reduces re-convergence waiting by executing both diverging paths and adaptively issuing threads waiting at re-convergence threads. DWS benefits from issuing the waiting threads if these threads end up in prefetching data for threads of the same warp. On the negative side, these threads may pass the re-convergence point at lower lane activity, compared to SBR. Compared to DWF, DWS provides better TLP enhancement as it exploits mechanisms to re-converge the diverged threads. TBC and LWM focus on lane activity without taking into account the impact on serialization and re-convergence waiting. Both solutions employ a large stack to maximize lane activity. This comes at the cost of exacerbated serialization and re-convergence waiting. The performance cost of this approach is little in the presence of multiple concurrent large warps. CAPRI significantly reduces the re-convergence waiting associated with TBC. However, and as reported by the authors, it suffers from higher re-convergence waiting compared to the SBR [Rhu and Erez 2012]. Moreover, CAPRI does not address diverging path serialization. SBI+SWI simultaneously merges and co-issues two narrow-occupied warps over SIMD. This method improves width utilization and eliminates diverging path serialization. The number of merging warps, however, is limited to two [Brunie et al. 2012]. DPE targets depth utilization and activates two parallel diverging paths. The mechanism effectively interleaves the execution of diverging paths to mitigate the stalled threads. DPE, however, limits the interleaving potential. Due to stack limitations, DPE can only interleave the diverging paths corresponding to the same branch. Accordingly, stalled threads under nested diverging paths cannot be interleaved. DPE does not impact the pipeline width utilization.

HARP can be used on top of [Fung and Aamodt 2011] and [Diamos et al. 2011] to possibly achieve higher performance by re-converging the threads earlier than the immediate post-dominator. Limiting re-convergence to the immediate post-dominator (as currently done by HARP) increases the number of regrouping opportunities at the expense of a later re-convergence. HARP is different from [Fung et al. 2007] as we guarantee returning to pre-divergence SIMD by re-converging at the immediate post-dominator. This eliminates undesirable consequences associated with [Fung et al. 2007] such as starvation eddies scenarios.

HARP is different from DWS [Meng et al. 2010] in two ways. First, HARP enhances lane activity by regrouping threads from different warps into the same

Table 3. Baseline configurations for GPGPU-sim. LWM's warp size is 128. HARP's warp size is 16 (equal to SIMD width).

NoC	
#SMs : #Memory Ctrls	16 : 6
#SM Sharing a Network Interface	4
Clocking	
Core : Interconnect : DRAM	1150 : 650 : 1500 MHz
Memory	
L2 Unified Cache per Memory Ctrls	128 KB : 16-way : LRU : 64Byte blocks
#Banks Per Memory Ctrls	8
DRAM Scheduling Policy : Queue Size	FCFS:32
GDDR5 memory timing	tRRD=5, tRCD=12, tRAS=28, tRP=12, tRC=35, tCL=10
SM	
# warp schedulers	2
#threads per warp scheduler	768
#SIMD groups : SIMD width : Pipeline depth	2 : 16 : 8
Maximum thread-blocks	16
Warp size	32
Register file : Shared memory	128KB : 32KB
L1 Data $	48KB : 12-way : LRU : 64Byte blocks
L1 Texture $	8KB : 2-way : LRU : 64Byte blocks
L1 Constant $	8KB : 2-way : LRU : 64Byte blocks

warp. Second, HARP reduces re-convergence waiting using small warps and with a workload independent approach. Using small warps synchronizes fewer threads reducing the re-convergence waiting. In addition, independent schedulable warps hide the latency of each other and can potentially act as a pre-fetcher for other warps. Whether the decisions made by the subdivision schemes improve performance depends on application prefetching frequency and is workload dependent.

Our work is different from TBC [Fung and Aamodt 2011] and LWM [Narasiman et al. 2011] in three ways. First, HARP eliminates diverging path serialization. Large re-convergence stack, as used by TBC and LWM, can exacerbate serialization since it may increase the number of inactive threads. HARP does not come with this drawback. Second, HARP improves re-convergence waiting using small warps. Larger stack synchronizes a larger number of threads at the re-convergence point, which can increase re-convergence waiting. HARP does not suffer from this increase as it limits synchronization to a smaller group of threads. Third, by focusing on short-life warps within a basic block, HARP does not require dealing with the complexities associated with SBR approaches maintaining a large number of diverged threads.

The key difference between HARP and CAPRI is that HARP does not inactivate threads due to divergence. Intuitively we expect CAPRI to have higher lane activity compared to HARP (similar to LWM). CAPRI takes a different approach to address re-convergence waiting and relies on predictor accuracy to reduce re-convergence waiting. As reported by the authors in [Rhu and Erez 2012], it is hard to predict compaction-adequacy accurately under complex control flow graphs leading to the performance near to TBC (e.g. MU).

HARP differs from SBI+SWI [Brunie et al. 2012] in several ways. First, SBI+SWI selects and co-issues diverged threads every cycle. HARP, however, regroups diverged threads into new warps and maintains the new warps until they diverge again or reach the re-convergence point. Second, while SBI+SWI executes a particular static instruction to ensure re-convergence, HARP synchronizes the diverged threads using dynamic barriers. Using static instruction to coordinate

Table 4. HARP's configuration details under GPGPU-sim.

SM	
#warp schedulers	2
#threads per warp scheduler	768
#entry in Second-level table per warp scheduler	512
#entry in Ready-Lookup table per warp scheduler	8
#entry in Waiting-Lookup table per warp scheduler	8
#entry in Re-convergence Barriers table per warp scheduler	128
#way in Re-convergence Barriers table per warp scheduler	16
#SIMD groups : SIMD width : Pipeline depth	2 : 16 : 8
Maximum thread-blocks	16
Warp size	16
Register file : Shared memory	128KB : 32KB
L1 Data $	48KB : 12-way : LRU : 64Byte blocks
L1 Texture $	8KB : 2-way : LRU : 64Byte blocks
L1 Constant $	8KB : 2-way : LRU : 64Byte blocks

warp-splits limits the scope of the solution to a single warp. In HARP, however, using dynamic barriers relaxes this restriction allowing multiple warps to share barriers. Finally, HARP allows higher number of diverged warps to be merged into one. Under SBI+SWI, the number of warps (or diverging paths) that can be merged is equal to the number of warp schedulers per SM. HARP does not limit this to the number of available warp schedulers. Instead, HARP temporarily stalls the diverged threads in a table to merge multiple narrowly occupied, recently diverged warps. It should be noted that HARP can be employed on top of the dual-instruction SIMD design suggested by Brunie et al. [2012] to co-issue two warps simultaneously.

There are two main differences between HARP and DPE. First, HARP targets width underutilization and regroups the diverged threads into new warps to improve lane activity. Second, HARP activates *all* diverging paths to unchain inactive threads and to minimize the stalled TLP, enhancing depth utilization.

6. METHODOLOGY

6.1 Simulator Infrastructure

We modified GPGPU-sim [Bakhoda et al. 2009] (version 2.1.1b) to evaluate HARP. We simulated the system shown in Table 3 to model the baseline microarchitecture described in Section 2. We modified GPGPU-sim to model a baseline GPU similar to NVIDIA Fermi [Wittenbrink et al. 2011]. We model two independent warp schedulers per SM issuing instructions on two SIMD groups backend. We have configured the GPGPU-sim to model GDDR5 timing of hynix H5GQ1H24AFR [hynix 2009] (as suggested by GPGPU-sim v3). We report performance in terms of throughput or IPC (instruction per clock). To estimate HARP's area overhead we use CACTI 6.5 [Muralimanohart et al. 2007].

We compare HARP to SBR, DWF [Fung et al. 2007], and LWM [Narasiman et al. 2011]. We configured DWF with 32 threads per warp, 256 entries per LUT and Majority issue heuristic [Fung et al. 2007]. We assume 128 threads per warp for LWM.

We assumed 512-entry Second-Level table, an 8-entry Ready-Lookup table, an 8-entry Waiting-Lookup table, and a 16-way 128-entry Re-convergence Barriers table for the baseline configuration in HARP. Table 4 summarizes HARP's configuration details.

Table 5. Benchmarks Characteristics. Concurrent threads per SM is twice the number of concurrent threads/scheduler.

Type	Benchmark name	Abbrev.	Grid Size	Block Size	Total dynamic instructions	Concurrent thread-block/SM	Concurrent threads/ scheduler
Type A	MUMmer-GPU++	MP	(1)	(256)	0.3M	1	256
	MUMmer-GPU	MU	(1)	(100)	0.2M	1	100
	N-Queen	NQU	(256)	(96)	1.2M	1	96
	Needleman-Wunsch	NW	2x(1) … 2x(127) (128)	255x(16)	207M	3/4	48/64
	Particle Filter	PF	9x(8)	9x(128)	42M	1	128
Type B	BFS Graph	BFS	16x(8)	16x(512)	1.4M	1	512
	Back Propagation	BKP	2x(1,64)	2x(16,16)	2.9M	4	512
	Hotspot	HSPT	(43,43)	(16,16)	76M	2	512
	Laplace 3D	LPS	(4,25)	(32,4)	81M	3	384
	Ray Tracing	RAY	(16,32)	(16,8)	65M	3	384
Type C	Coulomb Potential	CP	(8,32)	(16,8)	113M	8	768
	Fast Walsh Transform	FWAL	6x(32) 3x(16) 1x(128)	6x(256) 3x(512) 1x(512)	12M	1	256/512
	Matrix Multiply	MTM	(5,8)	(16,16)	2.4M	3	256

6.2 Benchmark Analysis

We use benchmarks from Rodinia [Che et al. 2009], CUDA SDK 2.3 [NVIDIA Corp. 2008], Parboil [Stratton et al. 2012], and the benchmarks distributed with GPGPU-sim [Bakhoda et al 2009]. We include MUMmerGPU++ [Gharaibeh and Ripeanu 2010] third-party sequence alignment program. Our benchmark set includes 18 benchmarks. Table 5 shows the benchmarks and their characteristics. In this table Grid Size and Block Size columns represent a list of grid sizes and thread-block sizes that are launched by the benchmark. Generally, each size in the list is a represented by Kx (X,Y,Z) where X, Y, and Z denote the dimension sizes of grid/thread-block and K denotes the number of kernels launched by the benchmark. The table omits K, Y, and Z when their value is equal to one. The last two columns report the occupancy of kernels, sometimes separated by / (e.g. two different kernel occupancies in NW). We have modified PF to generate static random numbers across different runs. In the remainder of this section, we provide deeper analysis by investigating benchmarks in more detail.

We classify the benchmarks used in this study to three types. Benchmarks like BKP, BFS, HSPT, LPS, MU, MP, NQU, NW, PF, and RAY exhibit low lane activity and high amount of stalled threads. These benchmarks can potentially compensate 1) their lane activity loss by bringing threads from other warps into inactive lanes, and 2) their pipeline depth underutilization by using stalled threads. Lane activity is the common critical performance impacting issue in these workloads. On the other hand, existence of abundant parallel threads can mitigate the performance impact of activating the stalled threads. We categorize these benchmarks into Type A and Type B. *Type A* benchmarks have limited thread-level parallelism, benefiting from

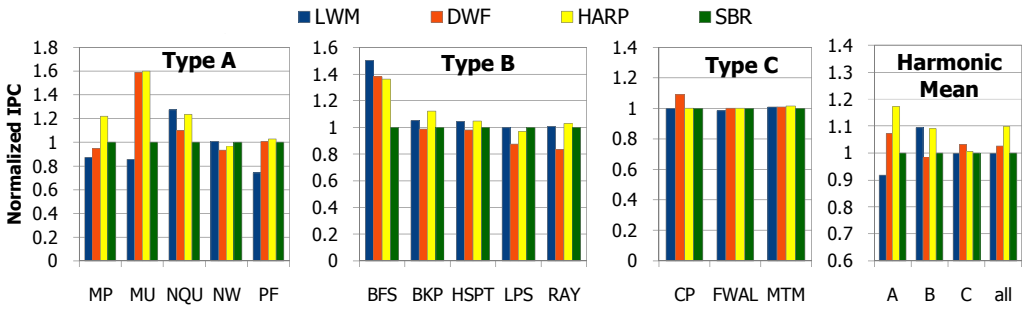

Figure 11. Performance for LWM, DWF, HARP, and SBR. The numbers are normalized to SBR.

activating the stalled threads (e.g. MP, MU, NQU, PF, and NW). *Type B* benchmarks have abundant thread-level parallelism, and are less likely to benefit from activating the stalled threads (e.g. BFS, BKP, LPS, HSPT, and RAY).

The remaining benchmarks, CP, FWAL, and MTM do not have stalled threads (due to inactivation or waiting at re-convergence). Consequently, these benchmarks lack the opportunity to improve pipeline depth utilization using stalled threads. Moreover, these benchmarks show full lane activity as there is no branch divergence. Therefore, there is no room to improve pipeline width utilization. In the light of these observations it is important to understand that we do not expect these benchmarks (and those with similar characteristics) to benefit from HARP. Nonetheless we include these benchmarks to represent applications losing performance mainly due to memory stalls. We refer to this group of applications as *Type C*.

7. EXPERIMENTAL RESULTS

In this section we present experimental results. In Section 7.1 we present performance, pipeline width/depth utilization, and thread-level parallelism analysis. In Section 7.2 we report sensitivity analysis. In Section 7.3 we report design space exploration for HARP. In Section 7.4 we discuss HARP's limitations.

7.1 In-depth Analysis

Performance. Figure 11 reports performance. We classify benchmarks according to Section 6.2. On average, HARP outperforms SBR, DWF, and LWM by 10% (max: 60%), 7% (max: 29%), and 10% (max: 88%), respectively.

HARP, expectedly, performs close to the baseline for Type C applications as this group of benchmarks do not face any of the challenges discussed in Section 3. For Type A and B, HARP improves performance by improving lane activity and TLP. This is achieved since HARP does not target TLP improvement (similar to DWF) or lane activity (similar to LWM) aggressively and in isolation. HARP improves both TLP and lane activity simultaneously to keep pipeline depth (TLP) and width (lane activity) utilized. For example, HARP performs the best for MP and MU while a previous solution, LWM, provides the best lane activity. This shows that maximizing lane activity is insufficient to achieve the best absolute performance.

Similarly, eliminating diverging path serialization and re-convergence waiting time without improving lane activity is not always successful. This is evidenced by RAY's performance loss under DWF. We conclude that our balanced solution can be more effective when both TLP and lane activity are critical.

LWM outperforms HARP for Type B applications having *enough parallel threads* to hide synchronization latency. The high TLP availability in this group of applications reduces the impact of stalled threads on performance. Accordingly, LWM

appears effective for Type B applications not suffering from re-convergence waiting and serialization (e.g., BFS and HSPT).

In the *absence of enough parallel threads* (Type A applications), HARP uses existing inactive/waiting parallelism to improve TLP and to keep the pipeline depth utilized. LWM, however, suffers from TLP reduction, which can out-weight its lane activity improvement (e.g., in MP, MU, and PF). Lack of enough parallelism is common for workloads employing large shared memory per thread-block or high number of register per threads [Volkov and Demmel 2008], which prevents running a high number of thread-blocks per SM.

In the remainder of this section, we analyze pipeline depth utilization, pipeline width utilization, and thread-level parallelism to provide better understanding regarding how HARP impacts performance.

Pipeline depth utilization. Figure 12 reports depths utilization for HARP and previous works. HARP enhances depth utilization in low-TLP Type A benchmarks by using their high number of stalled threads. For example, under MU and MP where the stalled threads are significant, depth utilization return is significant too. In NW, stalled threads are insignificant. Therefore, there is little room for depth utilization improvement under HARP. Type B benchmarks, however, already have high TLP along side high number of stalled threads. In these benchmarks, the depth utilization enhancement return achieved by HARP is not significant since the available TLP already provides high depth utilization. Finally, Type C benchmarks exhibit no stalled threads, leaving little depth utilization enhancement opportunity for HARP.

Pipeline width utilization. Figure 13 reports lane activity for different solutions. Type C applications show no variation in lane activity under LWM, DWF, and HARP performing close to SBR. Branch divergence does not happen in Type C applications leaving no room for improving SBR's lane activity. Type B applications, as reported earlier in Figure 4, exhibit a small gap between the width utilization of SBR and Oracle (except for BFS). According to Figure 13, even LWM, which merely focuses on width utilization enhancement, cannot improve width utilization significantly in Type B benchmarks. Type A benchmarks, however, show lane activity improvements under DWF, LWM, and HARP. DWF impacts lane activity at the expense of relaxing all re-convergences. For example, in NW, discarding re-convergence worsens lane activity under DWF as combinable-warps execute the same code block multiple times (notice DWF's depth utilization gain over other methods in Figure 12). HARP, on the other hand, employs dynamic re-convergence barriers on top of DWF to prevent such undesirable consequences. Therefore, HARP improves lane activity in MU, MP, NQU, BFS compared to SBR. HARP, however, reduces lane activity under LPS and RAY. For these benchmarks, Re-convergence Barriers table space limitation causes a few re-convergence barriers reservation misses. Subsequently, a few warps miss re-converging opportunities and continue to execute the re-convergence code block in multiple narrow-occupied warps. Larger Re-convergence Barrier table can mitigate this impact. Among the studied mechanisms, LWM achieves higher average lane activity compared to SBR, DWF, and HARP for Type A and B applications. LWM keeps many threads (here 128) at the same pace providing greater opportunity for regrouping and reducing inactive lanes. On the other hand, HARP and DWF do not produce new regrouping opportunities. *Achieving maximum lane activity alone does not necessarily translate to best performance when TLP is not high enough to fully utilize the pipeline depth. This is due to the fact that lane activity gains come with thread-level parallelism reduction and pipeline depth underutilization in the absence of enough TLP.*

Thread-Level Parallelism. Figure 14 reports the percentage of concurrent ready threads, which are stalled due to 1) synchronization at the re-convergence point, or 2) serialization of diverging paths in the re-convergence stack. As reported,

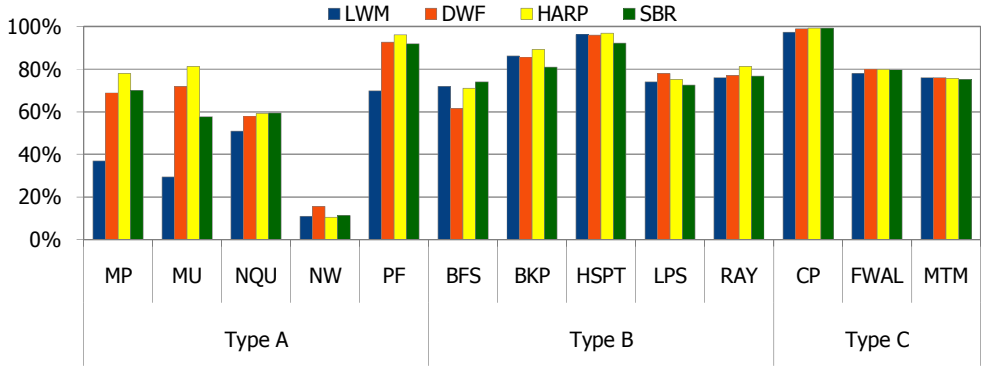

Figure 12. Pipeline depth utilization for LWM, DWF, HARP, and baseline (SBR).

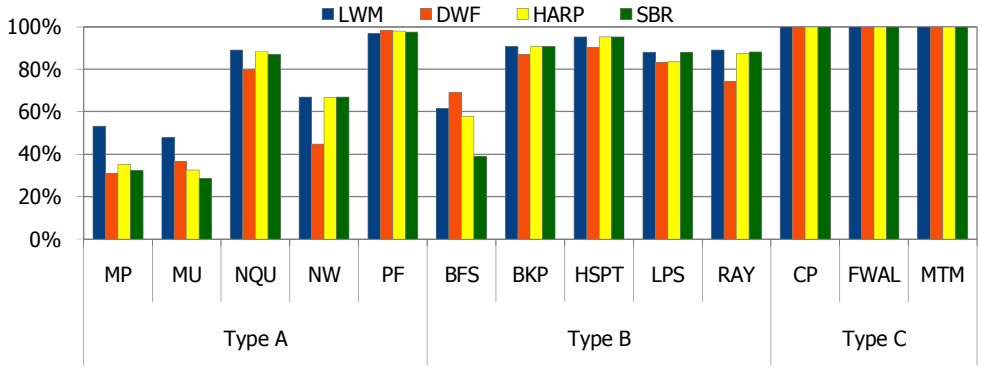

Figure 13. Pipeline width utilization for LWM, DWF, HARP, and baseline (SBR).

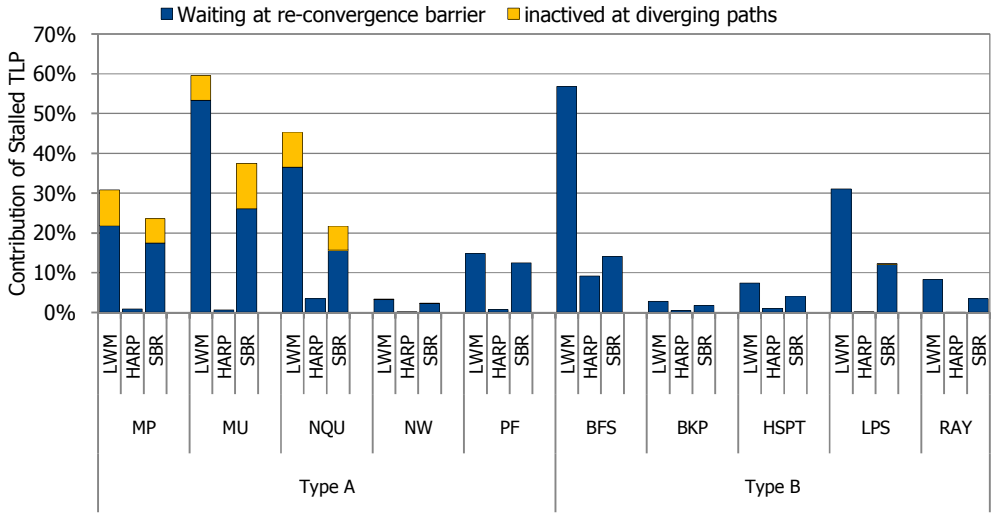

Figure 14. Share of threads waiting at the re-convergence barrier or inactivated at a diverging path during idle cycles of pipeline front-end. DWF eliminates re-convergence and serialization and therefore is not included. Numbers are normalized to SBR.

threads are stalled more frequently due to re-convergence synchronization than diverging path serialization. The share of stalled threads is significant in Type A and B applications. Type C applications are excluded since they exhibit no branch divergence (and zero stalled threads). LWM keeps many threads at the same pace increasing the number of stalled threads compared to SBR. HARP improves thread-level parallelism by reducing the number of stalled threads in two ways: 1) small

warps (as wide as SIMD width) are constructed to mitigate re-convergence waiting, and 2) diverged threads are grouped into new independent warps to eliminate serialization. TLP improvements translate to significant performance improvement if current active threads fail to utilize the entire pipeline depth (e.g. Type A applications).

7.2 Sensitivity Analysis

In this section we evaluate the sensitivity of our findings under different microarchitectural factors. Each factor may come with different levels. In order to analyze the performance of the system comprehensively, we require designing experiments that consider the combinations of all the levels of all factors. This is a formidable task in terms of computational complexity, since design space of the experiments grows exponentially with the number of factors and the number of levels of each factor. Below we use the theory of factorial design [Jain 1991] to obtain the performance of the system effectively by obtaining the maximum information with the minimum number of experiments. To this end, first we apply Plackett-Burman experimental design to find those factors that have higher impact on performance. Then, we re-evaluate our findings under variations in the reduced number of factors.

7.2.1 Plackett-Burman design

We apply two levels of Plackett-Burman (PB) design to find the most impacting factors. In the PB experiment there is one response, which depends on the variations of K independent factors. In the full design space, each factor can take two levels (low or high) leading to 2^K different experiments. PB design uses the theory of 2^k factorial design [Jain 1991] to find the most impacting factors through K+1 experiments.

We assume the relative throughput of HARP to SBR as the response and assume the following microachitectural parameters as independent factors: 1) pipeline depth, 2) L1 data cache size, 3) L1 Texture cache size, 4) L1 Constant cache size, 5) L2 unified cache size, 6) SIMD width, and 7) multithreading depth. For each factor, the low value represents the baseline configuration (Table 3) and the high value is assumed to be twice the low. For multithreading depth, we scale threads per SM, registers per SM, and shared memory per SM together. For L1 and L2 caches, we scale the number of sets in cache.

Table 6 shows the 8-run PB design in which each row is one experiment where the factors take low (-1) or high (1) value. The effect of each factor is calculated by summing the multiplication of the factor and response in each run divided by 4:

$$\text{Effect}(F_m) = \sum_{run=1}^{8} \frac{1}{4}(F_m[run]\text{Response}[run])$$

According to the effects, factor 1 (pipeline depth) and factor 6 (SIMD width) have the highest impact on HARP's speedup. In the remainder of this section, we focus on analyzing the sensitivity of HARP to these factors.

7.2.2 In-depth analysis

In order to study how system variations impact HARP, we focus on parameter changes with the highest impact: 1) deeper pipelines, and 2) wider SIMD widths. Figure 15 reports performance for different pipeline depths, and SIMD widths.

Pipeline depth. Conventional SIMT accelerators from AMD (e.g., Cayman), employ an 8-cycle latency pipeline [AMD, Inc. 2011] (at 800 MHz core clock). Recent NVIDIA GPUs take a similar approach to reduce core clock frequency [NVIDIA Corp. 2012b]. Previous NVIDIA accelerators, however, used higher frequencies (1.5 GHz). A previous work [Wong et al. 2010] showed that NVIDIA GT200 GPU pipeline

Table 6. Plackett-Burman experimental design. Response is the relative performance of HARP to SBR. Factors are pipeline depth (1), L1$ data size (2), L1$ Texture size (3), L1 Constant size (4), L2 cache size (5), SIMD width (6), and multithreading depth (7).

	Factors							Response
	F1	F2	F3	F4	F5	F6	F7	speedup
Run 1	1	-1	-1	1	-1	1	1	1.03
Run 2	1	1	-1	-1	1	-1	1	1.09
Run 3	1	1	1	-1	-1	1	-1	1.01
Run 4	-1	1	1	1	-1	-1	1	1.09
Run 5	1	-1	1	1	1	-1	-1	1.10
Run 6	-1	1	-1	1	1	1	-1	0.96
Run 7	-1	-1	1	-1	1	1	1	0.97
Run 8	-1	-1	-1	-1	-1	-1	-1	1.10
Effect	0.0283	-0.0100	-0.0020	0.0060	-0.0270	-0.1031	0.0007	

latency is 24 cycles for most instructions. Our baseline architecture models an 8-stage pipeline for all types of operations. To evaluate our solutions under an alternative pipeline, we also investigate a 16-cycle pipeline and report performance in Figure 15(a). As presented in 15(a), deep pipelining lowers IPC as the number of under-process warps in the pipeline increases. Deep pipelining reduces the gap between HARP and SBR. The deeper the pipeline, the higher the number of warps that can reside in different pipeline stages. This leaves less number of warps in the Lookup table. Under deeper pipelines, HARP has limited choices to merge warps compared to accelerators using shorter pipelines. Therefore, increasing the pipeline depth to 16 reduces average performance gap between HARP and SBR by 1% and 8% for Type A and Type B applications, respectively.

SIMD width. GPUs from different design spaces employ different SIMD widths. High-performance GPGPUs, for example, use large SIMD widths to reduce the associated data path overhead of many SIMD groups. While Tesla GPUs [Lindholm et al. 2008] employ one 8-wide SIMD group per SM, Fermi [Wittenbrink et al. 2011] has two 16-wide SIMD groups per SM and Kepler has 6 32-wide SIMD groups per SM [NVIDIA Corp. 2012b]. On the other hand, GPGPUs in smartphones and embedded devices employ narrower SIMDs. For example, PowerVR Series 5 OpenCL-capable GPUs exploit 4-wide SIMD [Imagination Technologies s2012]. To accommodate both spaces, we evaluate HARP under 8-wide and 32-wide SIMD, while maintaining warp size for the baseline SBR. As presented in Figure 15(b), wider SIMD reduces the gap between SBR and HARP. 8-wide SIMD increases the gap between HARP and SBR by 9%. Under 32-wide SIMD, SBR outperforms HARP by 4%. We have observed an increase in the re-convergence waiting under HARP for larger SIMD widths. This is due to the fact that many threads are locked together at common barriers. Particularly, under wider SIMD, HARP can construct wider barriers interlocking more threads. Consequently, the threads may experience longer waiting times at the re-convergence point. This waiting deprives the scheduler from ready threads, which can harm depth utilization. In fact, our study under wider SIMD shows that HARP can improve the width utilization significantly using larger barriers. However, the cost of this gain is depriving the pipeline from active/ready threads; degrading depth utilization. The depth utilization degradation can be compensated in Type B and C benchmarks with the vast available TLP. However, depth utilization degradation drops the performance significantly in the Type A benchmarks.

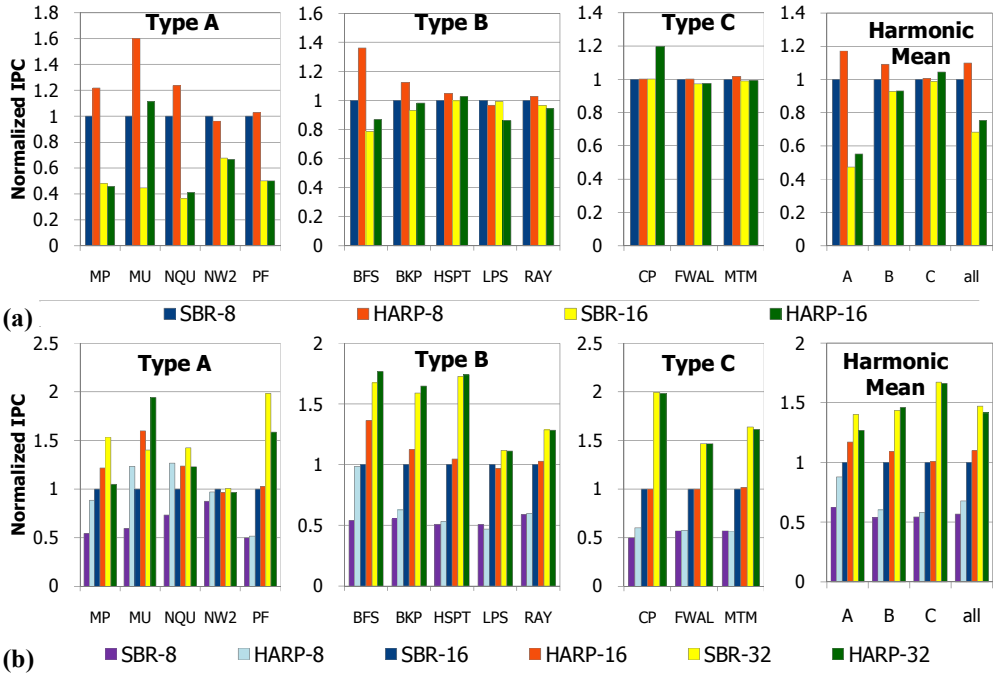

Figure 15. Performance for SBR and HARP for different pipeline depths and SIMD width. Sensitivity to (a) pipeline depth, and (b) SIMD width. Numbers are normalized to SBR 8-stage pipeline and 16-wide SIMD.

7.3 Design Space Exploration

HARP requires employing four auxiliary tables: Second-Level, Ready-Lookup, Waiting-Lookup, and Re-convergence Barriers tables. The size and structure of these tables impact HARP. In this study, and after testing many alternatives to work along our 768-thread warp schedulers, we selected a 512-entry Second-Level table, an 8-entry Ready-Lookup table, an 8-entry Waiting-Lookup table, and a 16-way 128-entry Re-convergence Barriers table. Our evaluation shows that a 512-entry Second-Level is large enough to avoid an overflow. Also, reducing the size of Ready-Lookup and Waiting-Lookup tables from eight to four comes with a performance cost of 2% and 6%, respectively.

While the above configuration works effectively for applications studied here, it is important to investigate potential overflow scenarios caused by applications requiring larger structures. We assume that upon an overflow in Second-Level, extra warps can be moved to the off-chip DRAM. Upon an overflow in Lookup tables, additional diverged warps are bypassed to the Second-Level. Upon an overflow in Re-convergence Barriers table, diverged threads fail to reserve a barrier and subsequently miss the rejoining chance at the re-convergence point. Below we evaluate the sensitivity to the Re-convergence Barriers table size followed by HARP timing overhead considerations.

7.3.1 Re-convergence Barriers Table Size

We assumed a 16-way set-associative table for the Re-convergence Barriers table. As discussed earlier (Section 4.2.2), every 16 threads having consequent thread identifiers map to the same set in the table and share 16 available barriers. For example if the table has two sets, threads ranging from T0 to T15 find their barriers in set0. Similarly, T17 to T32 find their barriers in set1, T33 to T48 find their barrier

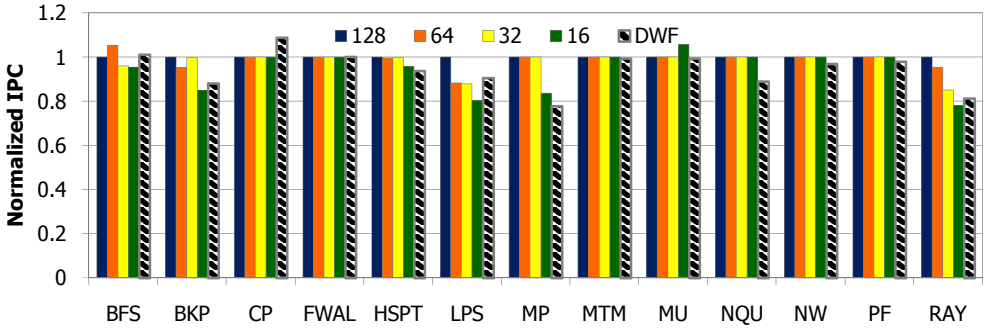

Figure 16. Performance sensitivity to the number of entries in the Re-convergence Barriers table.

in set0, and so on. Threads mapping to the same set compete for reserving their barrier. Changes in associativity and/or the number of sets can increase or reduce contention. Figure 16 reports the performance gain achieved by increasing the table size from 16 (one set) to 32 (two sets), 64 (four sets), and 128 (eight sets) while set associativity is 16. The bar on the far right reports performance for DWF. For the workloads with very limited branch divergence (e.g. CP, FWAL, HSPT, and MTM) or a few concurrent threads per warp scheduler (e.g. MU, MP, NQU, NW, and PF), a 32-entry (two sets) table is large enough to keep all dynamic re-convergence barriers. Reducing the table size below this number degrades performance due to an increase in the number of re-convergence misses. For some workloads (i.e., BFS, BKP, LPS, and RAY), tables with more than 32 entries are needed to prevent significant performance loss.

7.3.2 Timing Overhead of Re-convergence Barriers Table

The proposed microarchitecture for HARP performs searches and transitions. In this section we discuss the timing overhead of these operations and evaluate the performance impact under different timing scenarios.

Searches include searching Lookup tables upon divergence and searching Re-convergence Barriers table upon committing an instruction. These searches can be performed in parallel at the writeback stage. According to CACTI estimations, Lookup tables take 0.38 ns (0.18 ns to access the tag and 0.20 ns to access data) to access and therefore can be searched within a single cycle. We assume different performance/timing overhead alternatives for the Re-convergence Barriers table searches. According to CACTI estimations, it takes 0.63 ns (0.17 ns to access the tags and 0.46 ns to access the data) to access this table. Such accesses may result in violating pipeline's timing restrictions. Under such a pessimistic scenario, Re-convergence Barriers table can be pipelined. Accordingly, while we assume 1-cycle delay for accessing 8-set Re-convergence Barriers, we also measure the performance impact of pipelining the table access into 2 or 4 stages. Our study shows that the performance loss associated with this overhead is 3% and 8% for 2 and 4 stage table pipelines, respectively.

Transitions include issuing warps from Lookup and Re-convergence Barriers to Second-Level, from Second-Level to First-Level, and from First-Level to Waiting-Lookup. These transitions can be performed in a single cycle. HARP arbitrates lookup-to-second and barrier-to-second transitions and assigns higher priority to barrier-to-second to let the Lookup warps stay in the table for a longer time.

Table 7. Hardware overhead of HARP microarchitecture per SM in 40 nm technology. The number of bits in the tag modules include program counter (32-bit) and thread identifiers of 16-thread (96-bit). 6-bit (10-4) is enough to identify the thread uniquely among 1024 threads because the lane of thread in the SIMD (16-wide) does not change (4 least-significant bits of thread identifier can be ignored).

		Module Spec.						Area (mm2)	
		#banks	#entries	#sets	#ways	Tag bits	Row Size	Tag	Data
Re-conv. Barriers	Lanes	16	128	8	16	(10-4)	(10-4)+1	0.017	0.058
	RPC	1	128	8	16	1+32	32	0.010	0.010
Second-Level		1	512	512	1		32+(16*(10-4))+2		0.024
Lookup Ready		1	8	1	8	32+(16*(10-4))	32+(16*(10-4))	0.004	0.018
Lookup Waiting		1	8	1	8	32+(16*(10-4))	32+(16*(10-4))	0.004	0.018
Total for 1 warp scheduler								0.1622	
Total for 1 SM (two warp schedulers)								0.3244	

7.4 Limitations

HARP benefits from improving both lane activity and TLP. HARP can be less effective once and if the lane activity becomes the dominating factor. For example, LWM is probably a better choice for accelerators and workloads where lane activity is the dominant source of performance loss (Type B).

We investigated if HARP can benefit from aggregating concurrent threads at the same pace in order to improve regrouping efficiency (and lane activity) inside diverging paths. We examined several priority warp schedulers to keep threads at the same pace. Our study showed that the waiting time overhead outweighs the lane activity improvement.

8. HARDWARE OVERHEAD

In this section we estimate the area overhead associated with HARP. We used CACTI 6.5 [Muralimanohart et al. 2007] for the evaluation. Our baseline architecture models the architecture of NVIDIA Fermi GF100 [Wittenbrink et al. 2011]. Hence, we obtain the area in 40 nm technology and compare to publicly available NVIDIA GF100 information [Wikipedia 2013]. The overhead of HARP depends on several design parameters; namely Second-Level table and Re-convergence Barriers table. In this section we estimate the area overhead of a HARP configuration using 512-entry Second-Level, 128-entry 16-way associative Re-convergence Barriers, 8-entry fully-associative Ready Lookup table, and 8-entry fully-associative Waiting Lookup table.

Table 7 reports the estimated tables' sizes. HARP imposes 0.16 mm^2 area overhead per warp scheduler. The overhead for 16 SMs is 5.19 mm^2. Our baseline architecture is similar to NVIDIA Fermi GF100 which is manufactured on 529 mm^2 chip [Wikipedia 2013]. We estimate that HARP imposes 1% (5.19 divided by 529) area overhead for a Fermi-like architecture.

Under HARP, threads of the same warp may need to access different offsets in register banks. Previous studies [Fung et al. 2009; Narasiman et al. 2011] have introduced low cost solutions to address this problem. HARP takes a similar approach and associates one logical circuit with each bank to retrieve the register offset at each lane independently and in parallel. Previous studies [Fung et al. 2009; Narasiman et al. 2011] have estimated this area overhead to be about 2.5%.

9. CONCLUSIONS

In this study we analyzed branch divergence under conventional SIMT accelerators and found that a noticeable amount of concurrent threads stay inactive/waiting while their context is maintained in the register file. We introduced HARP to eliminate inactive threads, reduce the number of waiting threads and improve SIMD efficiency.

HARP is an innovative control-flow mechanism, which terminates, reconstructs and reincarnates short-life warps. The reconstructed warps enhance SIMD efficiency and reduce the impact of re-convergence waiting and divergence serialization by forming new and smaller warps using threads from different execution paths. We reported experiments confirming our motivating observations and compare HARP to conventional and alternative control-flow mechanisms. Moreover we studied how changes in system configurations impact overall results. We reported on hardware complexity and the overhead associated with the proposed design.

Acknowledgments

We thank anonymous reviewers for their valuable comments. We also thank Ali Shafiee for his comments on this work. This work was supported by School of Computer Science at the Institute for Research in Fundamental Sciences (IPM) and the Natural Sciences and Engineering Research Council of Canada, Discovery Grants Program.

REFERENCES

AMD Inc. 2012. AMD Accelerated Parallel Processing OpenCL.

Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2009)*. IEEE, 163-174.

Nicolas Brunie, Sylvain Collange, and Gregory Diamos. 2012. Simultaneous branch and warp interweaving for sustained GPU performance. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA '12)*. IEEE Computer Society, Washington, DC, USA, 49-60.

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 IEEE International Symposium on In Workload Characterization (IISWC 2009)*. IEEE, 44-54.

Sylvain Collange. 2011. Stack-less SIMT reconvergence at low cost. Technical Report. Available: http://hal.archives-ouvertes.fr/docs/00/62/26/54/PDF/collange_sympa2011_en.pdf

Gregory Diamos, Benjamin Ashbaugh, Subramaniam Maiyuran, Andrew Kerr, Haicheng Wu, and Sudhakar Yalamanchili. 2011. SIMD re-convergence at thread frontiers. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture* (MICRO-44 '11). ACM, New York, NY, USA, 477-488. DOI=10.1145/2155620.2155676 http://doi.acm.org/10.1145/2155620.2155676

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* (MICRO 40). IEEE Computer Society, Washington, DC, USA, 407-420. DOI=10.1109/MICRO.2007.12 http://dx.doi.org/10.1109/MICRO.2007.12

Wilson W. L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. 2009. Dynamic warp formation: Efficient MIMD control flow on SIMD graphics hardware. *ACM Trans. Archit. Code Optim.* 6, 2, Article 7 (July 2009), 37 pages. DOI=10.1145/1543753.1543756 http://doi.acm.org/10.1145/1543753.1543756

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 Computer Society, Washington, DC, USA, 25-36.

Abdullah Gharaibeh and Matei Ripeanu. 2010. Size Matters: Space/Time Tradeoffs to Improve GPGPU Applications Performance. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis* (SC '10). IEEE Computer Society, Washington, DC, USA, 1-12. DOI=10.1109/SC.2010.51 http://dx.doi.org/10.1109/SC.2010.51

Hynix Semiconductor. 2009. 1Gb (32Mx32) GDDR5 SGRAM H5GQ1H24AFR. Available: http://www.hynix.com/datasheet/pdf/graphics/H5GQ1H24AFR(Rev1.0).pdf

Imagination Technologies. 2012. PowerVR Series 5 architecture guide for developers. Available: http://www.imgtec.com/powervr/insider/docs/PowerVR%20Series%205.Architecture%20Guide%20for%20Developers.pdf

Raj Jain. 1991. The art of computer systems performance analysis (Vol. 182). John Wiley & Sons.

Adwait Jog, Onur Kayiran, Nachiappan Chidambaram Nachiappan, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R. Das. 2013. OWL: cooperative thread array aware scheduling techniques for improving GPGPU performance. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems* (ASPLOS '13). ACM, New York, NY, USA, 395-406. DOI=10.1145/2451116.2451158 http://doi.acm.org/10.1145/2451116.2451158

Khronos Group. 2013. OpenCL - The open standard for parallel programming of heterogeneous systems. Available: http://www.khronos.org/opencl/

Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. 2008. NVIDIA Tesla: A Unified

Graphics and Computing Architecture. *IEEE Micro* 28, 2 (March 2008), 39-55. DOI=10.1109/MM.2008.31 http://dx.doi.org/10.1109/MM.2008.31

Roberto Mijat. 2012. Take GPU Processing Power Beyond Graphics with Mali GPU Computing. Available: http://malideveloper.arm.com/downloads/WhitePaper_GPU_Computing_on_Mali.pdf

Jiayuan Meng, David Tarjan, and Kevin Skadron. 2010. Dynamic warp subdivision for integrated branch and memory divergence tolerance. *SIGARCH Comput. Archit. News* 38, 3 (June 2010), 235-246. DOI=10.1145/1816038.1815992 http://doi.acm.org/10.1145/1816038.1815992

Naveen Muralimanohar, Rajeev Balasubramonian, and Norm Jouppi. 2007. Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 40)*. IEEE Computer Society, Washington, DC, USA, 3-14. DOI=10.1109/MICRO.2007.30 http://dx.doi.org/10.1109/MICRO.2007.30

Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N. Patt. 2011. Improving GPU performance via large warps and two-level warp scheduling. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture* (MICRO-44 '11). ACM, New York, NY, USA, 308-317. DOI=10.1145/2155620.2155656 http://doi.acm.org/10.1145/2155620.2155656

NVIDIA Corp. 2008. NVIDIA CUDA SDK 2.3 [online]. Available: https://developer.nvidia.com/cuda-toolkit-23-downloads

NVIDIA Corp. 2012a. CUDA C Programming Guide [online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

NVIDIA Corp. 2012b. Kepler GK110 Architecture [online]. Available: http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

NVIDIA Corp. 2012c. CUDA C Programming Guide, Compute Capability section [online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/#compute-capabilities

NVIDIA Corp. 2012d. CUDA GPUs [online]. Available: https://developer.nvidia.com/cuda-gpus

Minsoo Rhu and Mattan Erez. 2012. CAPRI: prediction of compaction-adequacy for handling control-divergence in GPGPU architectures. In *Proceedings of the 39th Annual International Symposium on Computer Architecture* (ISCA '12). IEEE Computer Society, Washington, DC, USA, 61-71.

Minsoo Rhu and Mattan Erez. 2013. The dual-path execution model for efficient GPU control flow. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)* (HPCA '13). IEEE Computer Society, Washington, DC, USA, 591-602. DOI=10.1109/HPCA.2013.6522352 http://dx.doi.org/10.1109/HPCA.2013.6522352

John A. Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, Wen-mei W. Hwu. 2012. Parboil: A revised benchmark suite for scientific and commercial throughput computing. IMPACT Technical Report.

Vasily Volkov and James W. Demmel. 2008. Benchmarking GPUs to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing (SC '08)*. IEEE Press, Piscataway, NJ, USA, , Article 31 , 11 pages.

Wikipedia. 2013. GeForce 400 Series. Available: http://en.wikipedia.org/wiki/GeForce_400_Series

Craig M. Wittenbrink, Emmett Kilgariff, and Arjun Prabhu. 2011. Fermi GF100 GPU Architecture. *IEEE Micro* 31, 2 (March 2011), 50-59. DOI=10.1109/MM.2011.24 http://dx.doi.org/10.1109/MM.2011.24

Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. 2010. Demystifying GPU microarchitecture through microbenchmarking. In *Proceedings of IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS 2010)*. IEEE, 235-246.

Haicheng Wu, Gregory Diamos, Jin Wang, Si Li, and Sudhakar Yalamanchili. 2012. Characterization and transformation of unstructured control flow in bulk synchronous GPU applications. *Int. J. High Perform. Comput. Appl.* 26, 2 (May 2012), 170-185. DOI=10.1177/1094342011434814 http://dx.doi.org/10.1177/1094342011434814

Xuejun Yang, Xiangke Liao, Kai Lu, Qingfeng Hu, Junqiang Song, Jinshu Su. 2011. The TianHe-1A Supercomputer: Its Hardware and Software. *J. Comput. Sci. Technol. 26(3), 344-351.*

George L. Yuan, Ali Bakhoda, and Tor M. Aamodt. 2009. Complexity effective memory access scheduling for many-core accelerator architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture* (MICRO 42). ACM, New York, NY, USA, 34-44. DOI=10.1145/1669112.1669119 http://doi.acm.org/10.1145/1669112.1669119

Received December 2012; revised September 2013; accepted October 2013