# A Scalable Multi-Path Microarchitecture for Efficient GPU Control Flow

Ahmed ElTantawy[1], Jessica Wenjie Ma[1], Mike O'Connor[2], and Tor M. Aamodt[1]

[1]University of British Columbia
[2]NVIDIA Research

## Abstract

*Graphics processing units (GPUs) are increasingly used for non-graphics computing. However, applications with divergent control flow incur performance degradation on current GPUs. These GPUs implement the SIMT execution model by serializing the execution of different control flow paths encountered by a warp. This serialization can mask thread level parallelism among the scalar threads comprising a warp thus degrading performance. In this paper, we propose a novel branch divergence handling mechanism that enables interleaved execution of divergent paths within a warp while maintaining immediate postdominator reconvergence. This multi-path microarchitecture decouples divergence and reconvergence tracking by replacing the stack-based structure typically employed to support SIMT execution with two tables: a warp split table and a warp reconvergence table. It also enables reconvergence before the immediate postdominator which is important for efficient execution of unstructured control flow. Evaluated on a set of benchmarks with complex divergent control flow, our proposal achieves up to a 7× speedup with a harmonic mean of 32% over conventional single-path SIMT execution.*

## 1 Introduction

Current graphics processing units (GPUs) enable non-graphics computing using a Single Instruction Multiple Threads (SIMT) execution model. The SIMT model is typically implemented using a single instruction sequencer to operate on a group of threads, called a warp by NVIDIA, in lockstep. This amortizes instruction fetch and decode cost improving efficiency. To implement the SIMT model a mechanism is required to allow threads to follow different control flow paths. Current GPUs employ mechanisms that serialize the execution of divergent paths. Typically, they ensure reconvergence occurs at or before the immediate postdominator (IPDOM)[1] [9] of the divergent branch.

[1]The immediate postdominator of a branch is the earliest point in the program that all diverged threads from the branch are guaranteed to cross before the program exit.

This serialization of divergent control flow paths reduces thread level parallelism (TLP), but this reduction can be mitigated if the concurrent execution of divergent paths is enabled. Indeed, there have been several proposals to support multi-path execution in GPUs [17, 13, 4, 21]. However, previous multi-path proposals have limitations. Dynamic warp subdivision (DWS) [17] enables interleaving the execution of divergent control flow paths. DWS uses compiler heuristics to decide which branches start subdividing a warp into splits and which do not. Diverged splits reconverge at the IPDOM of a subdividing branch. However, the IPDOMs of branches nested within the subdividing branch are ignored. To compensate, the heuristics employed by DWS must carefully balance SIMD utilization with TLP. Dual-Path Stack (DPS) [21] enables interleaving two divergent paths while maintaining IPDOM reconvergence.

In this paper, we explore how to enable multi-path execution with support for an arbitrary number of divergent control flow paths while maintaining IPDOM reconvergence. We demonstrate that this is possible if the tracking of diverged control flow paths is decoupled from the tracking of their reconvergence points. Hence, we propose replacing the stack-based reconvergence mechanisms with two tables. One table tracks the concurrent executable paths, while the other tracks the reconvergence points. We demonstrate how to extend this mechanism to enable opportunistic early reconvergence to improve the SIMD unit utilization for applications with unstructured control flow behaviour [8]. Evaluated on a set of benchmarks with multi-path divergent control flow, our proposal achieves an average of 30% reduction in idle cycles, 48% average improvement in SIMD efficiency and 32% harmonic mean speedup compared to the conventional single-path execution. Also, our proposal achieves 22.5% harmonic mean speedup over DPS.

The contributions of this paper are:

- It proposes a new branch divergence handling mechanism that enables multi-path execution with support

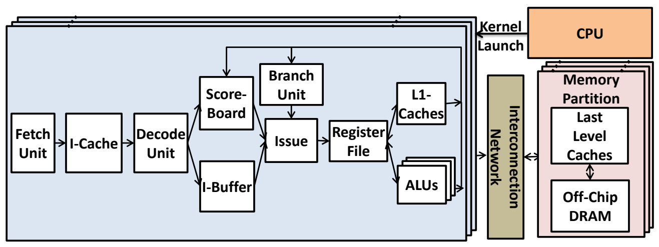


**Figure 1. Baseline architecture**

```
// id = thread ID
// BB_A Basic Block "A"
if(id%2==0){
    // BB_B
}else{
    // BB_C
}
// BB_D
```

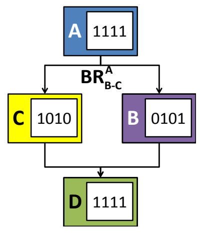
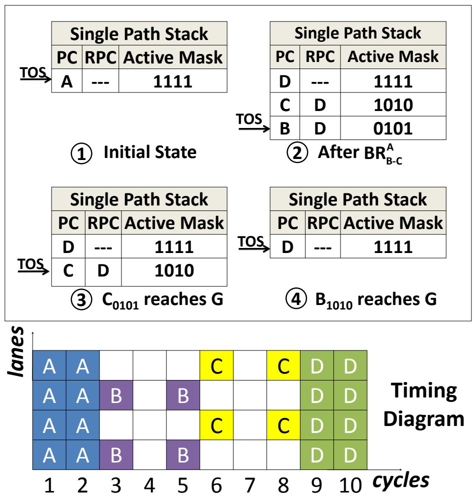


**Figure 2. Divergent code example**



① Initial State

| Single Path Stack | | |
|---|---|---|
| PC | RPC | Active Mask |
| A | --- | 1111 |

② After $BR^A_{B-C}$

| Single Path Stack | | |
|---|---|---|
| PC | RPC | Active Mask |
| D | --- | 1111 |
| C | D | 1010 |
| B | D | 0101 |

③ $C_{0101}$ reaches G

| Single Path Stack | | |
|---|---|---|
| PC | RPC | Active Mask |
| D | --- | 1111 |
| C | D | 1010 |

④ $B_{1010}$ reaches G

| Single Path Stack | | |
|---|---|---|
| PC | RPC | Active Mask |
| D | --- | 1111 |

**Figure 3. Execution with Single-Path Stack**

for an arbitrary number of diverged splits while maintaining reconvergence at the IPDOM.

- It extends the proposed mechanism to enable early reconvergence opportunistically at run-time.
- With detailed analysis, we show that our proposal provides up to a 7× speedup with a harmonic mean of 32% over the single-path execution model; and up to 4.5× speedup with a harmonic mean of 22.5% over the recent Dual-Path execution proposal [21].

The rest of this paper is organized as follows: Section 2 describes our baseline and the conventional stack-based reconvergence mechanism. Section 3 describes the multi-path microarchitecture. In Section 4, we extend it to enable opportunistic reconvergence at early reconvergence points. Section 5 describes our methodology. Section 6 describes experimental results, Section 7 summarizes related work and Section 8 concludes.

## 2 Baseline SIMT Accelerator

This section describes our baseline architecture and explains the branch divergence problem in GPUs.

### 2.1 Architecture

We study modifications to the GPU-like accelerator architecture shown in Figure 1. Current GPU designs consist of multiple processing cores. Each core consists of a set of parallel lanes (or SIMD units). Initially, an application begins execution on a host CPU, then a kernel is launched on the GPU in the form of a large number of logically independent scalar threads. These threads are split into logical groups operating in lockstep in a SIMD fashion (referred to as warps). Each SIMT core interleaves a number of warps on a cycle-by-cycle basis. The Instruction Buffer unit (I-Buffer) contains storage to hold decoded instructions and register dependency information for each warp. The scoreboard unit is used to detect register dependencies. A branch unit manages control flow divergence. The branch unit abstracts both the storage and the control logic required for divergence and reconvergence. Section 2.2 explains in detail the branch unit used in our baseline.

The issue logic selects a warp with a ready instruction in the instruction buffer to issue for execution. Based on the active mask of the warp, threads that should not execute, due to branch divergence, are disabled. The issued instructions fetch their operands from the register file. It is then executed on the corresponding pipeline (ALU or MEM).

### 2.2 Branch Divergence in GPUs

Figure 2 illustrates a simple example of divergent code and its corresponding control flow graph (CFG). The bit mask in each basic block of the CFG denotes which threads of a single warp containing four threads will execute that block. The rightmost bit represents thread with thread ID=0. All threads execute basic block A. Upon executing *divergent* branch $BR^A_{B-C}$, warp $A_{1111}$ diverges into two warp *splits* [17] $B_{0101}$ and $C_{1010}$. In our notation, branches are abbreviated as $BR$ with a superscript representing the basic block containing the branch and a subscript represents the successor basic blocks. Each warp split is represented by a letter representing the basic block that the split is executing with a subscript indicating the active threads.

The *immediate postdominator (IPDOM)* of the branch $BR^A_{B-C}$, basic block D, is the earliest point where all threads diverging at the branch are guaranteed to execute. We say an execution mechanism supports *IPDOM reconvergence* if it guarantees all threads in the warp that are active at any given branch are again active (executing in lockstep) when the immediate postdominator of that branch is next encountered. IPDOM reconvergence is favorable because the immediate postdominator is the closest point all threads in a warp are guaranteed to reconverge[1]. A mechanism for supporting IPDOM reconvegence using a stack of active masks [16] was introduced by Fung et al. [9]. However, there are different possible implementations that can support IPDOM reconvergence as defined above.

The postdominance relation between basic blocks can be represented in the form of a *postdominator tree*. In a postdominator tree, each node's descendants are those nodes it immediately dominates. We refer to basic blocks that are not ancestors or children with respect to each other as *parallel* basic blocks. Warp splits at parallel basic blocks can execute independently without impacting the reconvergence location. For example, warp splits $B_{1010}$ and $C_{0101}$ can execute independently until they reach basic block D [17].

**Single-Path Stack Execution Model:** Several designs have been proposed for handling branch divergence [16, 9, 25, 26]. They can achieve IPDOM recovergence but serialize execution of parallel control flow paths such that one split of a warp is scheduled repeatedly until reaching a reconvergence point.

Figure 3 illustrates the operation of the Single-Path Stack (SPS) model that we compare against in this paper, called IPDOM by Fung et al. [9], while it is executing the code example in Figure 2. The example assumes that each basic block contains two dependent instructions and that all instructions have one cycle latency, except the first instructions in blocks B and C, which have two cycle latency.

In SPS, a per-warp stack is used to manage divergent control flow. Initially, the stack has a single entry during the execution of basic block A ①. Once branch $BR^{A}_{B-C}$ is executed, the PC of the diverged entry is set to the reconvergence PC (RPC) of the branch (D). Also, resulting warp splits, $C_{1010}$ and $B_{0101}$, are pushed onto the stack ②. The RPC of the new entries is set to the RPC of the executed branch (D). At this point, only warp split $B_{0101}$ is eligible for scheduling, as it resides at the top of the stack (TOS entry). Warp split $C_{1010}$ is not at the top of the stack, hence, it cannot be scheduled. As a result, on cycle 4, there are no instructions to hide the latency of the first instruction in basic block B. Once warp split $B_{0101}$ reaches its reconvergence point (D), its corresponding entry is popped from the stack ③. Then, warp split $C_{1010}$ executes until it reaches its reconvergence point (D) after which it is popped from the stack ④. Finally, the diverged threads reconverge at $D_{1111}$. The above execution results in two idle cycles.

**Stack-Based Reconvergence Limitations:** The SPS execution model allows only a single control flow path to execute at a time, which reduces the number of running threads. Active threads on alternate paths that are not on the top of stack may be either waiting at a reconvergence point or ready to execute a parallel control flow path. During cycles when the pipeline is idle due to long latency events these alternate control paths could make progress, as observed by Meng et al. [17]. Thus, the SPS model captures only a fraction of the actual thread level parallelism (TLP). The remaining TLP is essentially masked by a structural hazard implicit in the use of a stack for implementing IPDOM reconvergence.

[1]Likely convergence [10] and thread frontiers [8] identify earlier reconvergence points that can occur dynamically in unstructured control flow if a subset of paths between branch and IPDOM are executed by a warp.

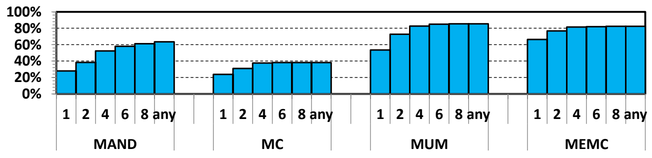


**Figure 4. Fraction of running scalar threads while varying maximum warp splits and assuming IPDOM reconvergence**

Figure 4 quantifies this by showing the amount of TLP available to the scheduler as we increase the maximum number of concurrently executable warp splits supported by hardware while IPDOM reconvergence is maintained. The graph plots the average portion among all the scalar threads that can be scheduled because they are active in the top entry of the stack. The SPS execution model corresponds to enabling a single warp split. Section 5 gives more details about the benchmarks and the methodology.

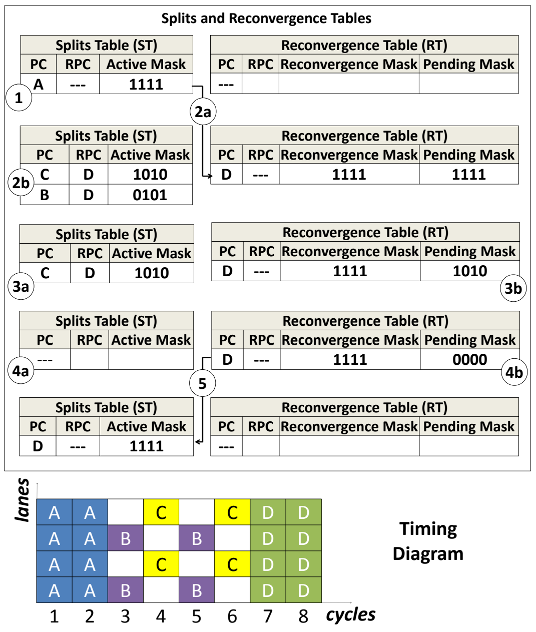
For this set of benchmarks, the SPS execution model captures from 15% of overall TLP in the ***MC*** benchmark up to around 65% in ***MEMC***. Figure 4 suggests that up to 35% more TLP is available when moving from SPS to a mechanism allowing any number of warp splits to be concurrently scheduled while maintaining IPDOM reconvergence. TLP does not go to 100% with unlimited warp splits because some threads need to wait at reconvergence points.

## 3 Multi-Path IPDOM (MP IPDOM)

In this section we propose a hardware mechanism that allows concurrent scheduling of any number of warp splits while still maintaining IPDOM reconvergence. To achieve this we replace the SIMT reconvergence stack structure with two tables. The warp Split Table (ST) records the state of warp splits executing in parallel basic blocks (i.e., blocks that do not dominate each other), which can therefore be scheduled concurrently. The Reconvergence Table (RT) records reconvergence points for the splits. The ST and RT tables work cooperatively to ensure splits executing parallel basic blocks will reconverge at IPDOM points.

### 3.1 Operation of MP IPDOM

We use the same simple control flow graph in Figure 2 to explain the high level operation of the Multi-Path IPDOM



| Splits Table (ST) | | |
|---|---|---|
| PC | RPC | Active Mask |
| A | --- | 1111 |

| Reconvergence Table (RT) | | | |
|---|---|---|---|
| PC | RPC | Reconvergence Mask | Pending Mask |
| --- | | | |

| Splits Table (ST) | | |
|---|---|---|
| PC | RPC | Active Mask |
| C | D | 1010 |
| B | D | 0101 |

| Reconvergence Table (RT) | | | |
|---|---|---|---|
| PC | RPC | Reconvergence Mask | Pending Mask |
| D | --- | 1111 | 1111 |

| Splits Table (ST) | | |
|---|---|---|
| PC | RPC | Active Mask |
| C | D | 1010 |

| Reconvergence Table (RT) | | | |
|---|---|---|---|
| PC | RPC | Reconvergence Mask | Pending Mask |
| D | --- | 1111 | 1010 |

| Splits Table (ST) | | |
|---|---|---|
| PC | RPC | Active Mask |
| --- | | |

| Reconvergence Table (RT) | | | |
|---|---|---|---|
| PC | RPC | Reconvergence Mask | Pending Mask |
| D | --- | 1111 | 0000 |

| Splits Table (ST) | | |
|---|---|---|
| PC | RPC | Active Mask |
| D | --- | 1111 |

| Reconvergence Table (RT) | | | |
|---|---|---|---|
| PC | RPC | Reconvergence Mask | Pending Mask |
| --- | | | |

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| A | A | | C | | C | D | D |
| A | A | B | | B | | D | D |
| A | A | | C | | C | D | D |
| A | A | B | | B | | D | D |

**Figure 5. Execution with Multi-Path IPDOM**

with the same assumptions described in Section 2.2. Figure 5 shows the operation of the MP IPDOM illustrating changes to the ST and RT tables (top) along with the resulting pipeline issue slots (bottom).

The warp begins executing at block A. Since there is no divergence, there is only a single entry in the ST, and the RT is empty ①. The warp is scheduled on the pipeline until it reaches the end of block A. After the warp executes branch $BR^{A}_{B-C}$ on cycle 2, warp $A_{1111}$ diverges into two splits $B_{0101}$ and $C_{1010}$. Then, the $A_{1111}$ entry is moved from the ST to the RT (2a) with PC field set to the RPC of branch $BR^{A}_{B-C}$ (i.e., D). The RPC can be determined at compile time and either conveyed using an additional instruction before the branch or encoded as part of the branch itself (current GPUs typically include additional instructions to manipulate the stack of active masks). The Reconvergence Mask entry is set to the same value of the active mask of the diverged warp split before the branch. The Pending Mask entry is used to represent threads that have not yet reached the reconvergence point. Hence, it is also initially set to the same value as the active mask. At the same time, two new entries are inserted into the ST; one for each side of the branch (2b). The active mask in each entry represents threads that execute the corresponding side of the branch.

On the clock cycle 3, warp splits $B_{0101}$ and $C_{1010}$ are eligible to be scheduled on the pipeline independently. We assume that the scheduler interleaves the available warp splits. Warp splits $B_{0101}$ and $C_{1010}$ hide each others' latency leaving no idle cycles (cycles 3-5). On cycle 6, warp split $B_{0101}$ reaches the reconvergence point (D) first. Therefore, its entry in the ST table is invalidated (3a), and its active mask is subtracted from the pending active mask of the corresponding entry in the RT table (3b). Later, on cycle 7, warp split $C_{1010}$ reaches reconvergence point (D). Thus, its entry in the ST table is also invalidated (4a), and its active mask is subtracted from the pending active mask of the corresponding entry in the RT table (4b). Upon each update to the pending active mask in the RT table, the Pending Mask is checked if it is all zeros, which is true in this case. The entry is then moved from the RT table to the ST table ⑤. Finally, the reconverged warp $D_{1111}$ executes basic block D on cycles 7 and 8.

## 3.2 Example with Nested Divergence

Figure 6 illustrates how MP IPDOM handles nested branches. It shows the state of the ST and RT tables after each step of executing the control flow graph in the left part of the figure. In our explanation of this example, we assume a particular sequence of events that results from one possible scheduling order. However, Multi-Path IPDOM does not require a specific scheduling order.

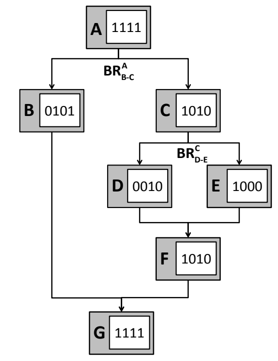
Initially, a single entry in the ST exists for warp split $A_{1111}$ ①. After branch $BR^{A}_{B-C}$, the MP control unit updates the ST and RT tables in a way identical to that described in Section 3.1 ②. Both warp splits in $B_{0101}$ and $C_{1010}$ are scheduled on the pipeline. Subsequently, warp split $C_{1010}$ diverges at $BR^{C}_{D-E}$ ③. Hence, the entry corresponding to $C_{1010}$ in the ST table is moved to the RT table with PC field set to the the reconvergence point of $BR^{C}_{D-E}$, which is F in this case. Also, two new entries corresponding to both sides of $BR^{C}_{D-E}$ are added to the ST table. As explained in detail in Section 3.3, the new entries are added to the first unallocated entries in the ST table.

At this point MP IPDOM exposes parallelism in three parallel control flow paths. Later, warp split $B_{0101}$ reaches the reconvergence point G ④. The MP control unit looks for the entry in the RT with PC=G and its active mask is a super set of the active mask of $B_{0101}$ entry in the ST table. As described in Section 3.3, the ST can store direct indices to the reconvergence entries in the RT. However, as later explained in Section 4, a performance optimized version of MP IPDOM uses an associative search through ST and RT table entries to detect earlier reconvergence opportunities.

The reconvergence entry is found to be $G_{1111}$. The active mask of warp split $B_{0101}$ is then subtracted from the Pending Mask of the reconvergence entry $G_{1111}$. Finally, the $B_{0101}$ entry in the ST table is invalidated ④.

Later warp split $E_{1000}$ reaches its reconvergence point F ⑤. The Pending Mask of the reconvergence entry $F_{1010}$ is updated accordingly, and the $E_{1000}$ entry in the ST table is invalidated. Then, warp split $D_{0010}$ reaches the same reconvergence point (F). The ST entry is removed and the reconvergence entry $F_{1010}$ is updated again to mark the arrival of warp split $E_{1000}$ ⑥.

```
// id = warp ID
// BB_A Basic Block "A"
if(id%2==0){
    // BB_B
}else{
    // BB_C
    if(id==1){
        // BB_D
    }else{
        // BB_E
    }
    // BB_F
}
// BB_G
```



(a) Code and CFG

| Splits Table (ST) | | |
|---|---|---|
| PC | RPC | Active Mask |
| A | --- | 1111 |

| Reconvergence Table (RT) | | | |
|---|---|---|---|
| PC | RPC | Rec. Mask | Pending Mask |
| --- | | | |

① Initial State

| Splits Table (ST) | | |
|---|---|---|
| PC | RPC | Active Mask |
| B | G | 0101 |
| C | G | 1010 |

| Reconvergence Table (RT) | | | |
|---|---|---|---|
| PC | RPC | Rec. Mask | Pending Mask |
| G | --- | 1111 | 1111 |

② After $BR^{A}_{B\text{-}C}$

| Splits Table (ST) | | |
|---|---|---|
| PC | RPC | Active Mask |
| B | G | 0101 |
| D | F | 0010 |
| E | F | 1000 |

| Reconvergence Table (RT) | | | |
|---|---|---|---|
| PC | RPC | Rec. Mask | Pending Mask |
| G | --- | 1111 | 1111 |
| F | G | 1010 | 1010 |

③ After $BR^{C}_{D\text{-}E}$

| Splits Table (ST) | | |
|---|---|---|
| PC | RPC | Active Mask |
| D | F | 0010 |
| E | F | 1000 |

| Reconvergence Table (RT) | | | |
|---|---|---|---|
| PC | RPC | Rec. Mask | Pending Mask |
| G | --- | 1111 | 1010 |
| F | G | 1010 | 1010 |

④ $B_{0101}$ reconverge at G

| Splits Table (ST) | | |
|---|---|---|
| PC | RPC | Active Mask |
| D | F | 0010 |

| Reconvergence Table (RT) | | | |
|---|---|---|---|
| PC | RPC | Rec. Mask | Pending Mask |
| G | --- | 1111 | 1010 |
| F | G | 1010 | 0010 |

⑤ $E_{1000}$ reconverge at F

| Splits Table (ST) | | |
|---|---|---|
| PC | RPC | Active Mask |
| --- | | |

| Reconvergence Table (RT) | | | |
|---|---|---|---|
| PC | RPC | Rec. Mask | Pending Mask |
| G | --- | 1111 | 1010 |
| F | G | 1010 | 0000 |

⑥ $D_{0010}$ reconverge at F

| Splits Table (ST) | | |
|---|---|---|
| PC | RPC | Active Mask |
| F | G | 1010 |

| Reconvergence Table (RT) | | | |
|---|---|---|---|
| PC | RPC | Rec. Mask | Pending Mask |
| G | --- | 1111 | 1010 |

⑦ $F_{1010}$ move to ST table

| Splits Table (ST) | | |
|---|---|---|
| PC | RPC | Active Mask |
| --- | | |

| Reconvergence Table (RT) | | | |
|---|---|---|---|
| PC | RPC | Rec. Mask | Pending Mask |
| G | --- | 1111 | 0000 |

⑧ $F_{1010}$ reconverge at G

| Splits Table (ST) | | |
|---|---|---|
| PC | RPC | Active Mask |
| G | -- | 1111 |

| Reconvergence Table (RT) | | | |
|---|---|---|---|
| PC | RPC | Rec. Mask | Pending Mask |
| --- | | | |

⑨ $G_{1111}$ move to ST table

(b) ST and RT tables (only valid entries shown)

**Figure 6. Example of Multi-Path IPDOM execution with nested divergence**

Upon updating the Pending Mask of the reconvergence entry $F_{1010}$, the MP control unit detects that there are no more pending threads for this entry (the Pending Mask is all zeros). Hence, the MP control unit moves the reconvergence entry to the ST table, setting the active mask to the Reconvergence Mask ⑦. Finally, warp split $F_{1010}$ reaches the reconvergence point G and updates the reconvergence entry $G_{1111}$. The control unit detects that the Pending Mask of entry $G_{1111}$ is all zeros ⑧ and moves the entry from the RT table to the ST table ⑨.

This example does not cover two special cases. The first case is when one side of the branch directly diverges to the reconvergence point of the branch (e.g., if clause with no else). In this case, the Pending Mask of the corresponding reconvergence entry is updated to mark this side of branch as converged, and there is no need to register a new entry for it in the ST. The second case is when a warp split encounters a branch whose reconvergence point is the same as the reconvergence point of the diverged warp split (e.g., backward branches that create loops). In such a case, there is no need to add a new entry to the RT table, since the corresponding reconvergence entry already exists.

## 3.3 Implementation

The modifications to the baseline pipeline consist of changes to three main parts: branch unit, instruction buffer, and issue unit. The changes are discussed in detail below.

### 3.3.1 Branch Unit and Instruction Buffer

Figure 7 shows the details of the branch unit and the instruction buffer (I-Buffer). Each warp has its own ST and RT tables. The PC entry in ST and RT tables points to the first instruction in a basic block. I-Buffer entries are allocated at the warp split granularity and modified to add the Split-ID, RPC and Active Mask entries. The I-Buffer is sized to hold a number of splits equal to the maximum number of warps per core. The maximum number of entries in the ST table equals the number of threads in a warp. The size of the RT table depends on the nesting depth of the application control flow. Section 6.6 presents the maximum usage of entries in both the ST and RT tables for our studied benchmarks. Similar to stack-based implementations, RT entries could be spilled to memory if the benchmark has a very large nesting depth [6].

Upon divergence, the branch control unit invalidates the associated split entry in the I-Buffer. Hence, it is no longer

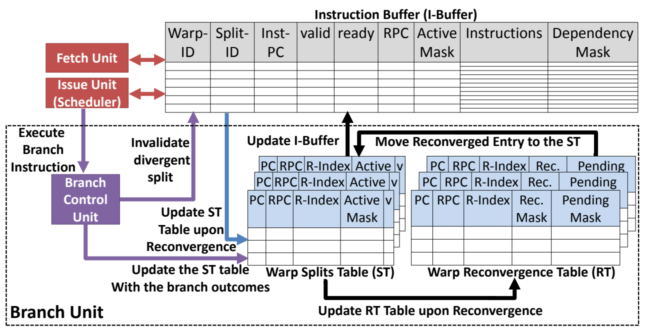
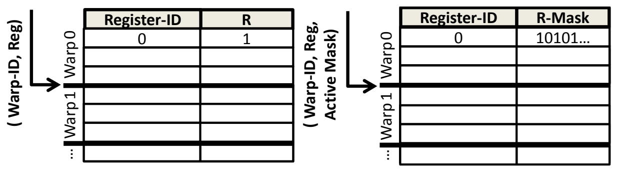

Instruction Buffer (I-Buffer)
Warp-ID
Split-ID
Inst-PC
valid
ready
RPC
Active Mask
Instructions
Dependency Mask
Fetch Unit
Issue Unit (Scheduler)
Execute Branch Instruction
Invalidate divergent split
Branch Control Unit
Update ST Table upon Reconvergence
Update the ST table With the branch outcomes
Update I-Buffer
Move Reconverged Entry to the ST
PC
RPC
R-Index
Active Mask
v
Rec. Mask
Pending Mask
Warp Splits Table (ST)
Warp Reconvergence Table (RT)
Branch Unit
Update RT Table upon Reconvergence

**Figure 7. Multi-Path IPDOM implementation**



(a) Single-Path IPDOM (b) Multi-Path IPDOM

**Figure 8. Changes required to the scoreboard logic to support Multi-Path IPDOM**

eligible for fetch or issue. The branch control unit marks the entry of the associated split in the ST table as unallocated. It also signals the ST table to transfer this entry to an unused entry in the RT table after modifying the PC field to the RPC of the branch. The index of this entry, R-Index, is stored, along with the new warp splits resulting from the divergence, in the ST table. The R-Index is used to directly index the corresponding reconvergence entry in the RT table upon reconvergence. Warp splits in the I-Buffer contain warp and split IDs. The split ID is an index identifying the warp split in the ST table. After a new warp split is created, it is written to a free I-Buffer entry. If there are more splits than I-Buffer entries, the extra splits sit idle in the ST table until an I-Buffer entry is free.

Upon reconvergence, the branch control unit invalidates the reconverged entry in both the I-Buffer and the ST table. It uses the R-Index field of the reconverging split to index the RT table to update the pending active mask as described in Section 3.1. Finally, the Pending Mask of the updated entry is checked; if it is all zeros, it moves the entry to the first unallocated entry in the ST table.

### 3.3.2 Scoreboard Logic

Current GPUs use a per-warp scoreboard to track data dependencies [7]. A form of set-associative look-up table (Figure 8a) is employed, where sets are indexed using warp ids and entries within each set contain a destination register ID of an instruction in flight for a given warp. When a new instruction is decoded, its source and destination register IDs are compared against the scoreboard entries of its warp. A dependency mask that represents registers that are causing data dependency hazards is produced from these comparisons and stored in the I-Buffer with the decoded instruction. The dependency mask is used by the scheduler to decide the eligible instructions at each issue slot. An instruction is eligible only if its dependency mask is all zeros. After writeback, both the scoreboard and the I-Buffer entries are updated to mark the dependency as cleared. In particular, the entries in the scoreboard look-up table that correspond to the destination registers of the written-back instruction are invalidated, and the bits that correspond to these destination registers in the dependency mask are cleared for all decoded instructions from this warp.

The Multi-Path IPDOM supports multiple number of concurrent warp splits running through parallel control flow paths. Hence, it is essential for the scoreboard logic to correctly handle dependencies for all warp splits and across divergence and reconvergence points. It is also desirable for the scoreboard to avoid declaring a dependency exists when a read in one warp split follows a write to the same register but from another warp split.

Therefore, we modify the scoreboard design by adding a reserved mask (R-mask) field to each entry in the scoreboard look-up table as shown in Figure 8b. When an instruction is scheduled it bitwise-ORs the R-mask of its destination register with its active mask. When a new instruction is decoded, its source and destination register IDs are compared against the scoreboard entries of its warp. If the R-mask bit of a register operand of the instruction is set for any of the instruction's active threads then a bit is set in the dependency mask for the I-Buffer entry associated with the instruction. This means that there is a pending write to this register by at least one active thread.

Upon writeback, the destination register's dependencies are cleared from the scoreboard by clearing the bits in the R-mask that correspond to the active threads in the written-back instruction. The dependency bit masks in the I-Buffer are also updated. To do so, the active mask of each instruction that belongs to the written-back warp is compared with the R-mask of the destination registers of the written-back instruction. The respective destination register bit in the dependency mask is cleared if the instruction active mask and the R-mask do not have any common active bits. An instruction in the I-Buffer is available to be issued if all the bits in the dependency mask are zero.

To illustrate the operation of the modified scoreboard and its interactions with the I-Buffer, we use the example code snippet in Figure 9. Initially, the scoreboard is empty and $I_0$ has no pending dependencies with any registers. At ①, $I_0$ is issued and it reserves a scoreboard entry for its destination register R0 with R-mask=1111. Also, the branch instruction splits the warp into two splits that fetch and decode instructions from the two sides of the branch (i.e., $I_1$ and $I_2$). At the decode stage, all the source and destination registers of the decoded instruction are checked against the reserved registers in the scoreboard unit to see if there are any pending dependencies, and, accordingly, the dependency mask of the instruction is generated. In this example, $I_1$ is dependent on R0 while $I_2$ has no pending dependencies. Hence, $I_2$ is eligible to be issued, and once it is issued, at ②, it reserves R1 with an R-mask=1010. At ③, $I_3$ is fetched and decoded. $I_3$ has pending dependencies on both R0 and R1, hence its Dep-mask=11. At ④, $I_0$ writes the load value to R0 and hence it releases R0 and clears the R-mask of the R0 entry from the scoreboard unit (the R0 entry becomes invalid since its R-mask is all zeros). Also, it clears the dependency mask of both $I_1$ and $I_3$ since the R-mask of R0 is now zeros for all active threads of both instructions. Since the dependency mask of $I_1$ is all zeros, it becomes eligible to be issued. Hence, at ⑤, it reserves its destination register R1 with an R-mask=1010; such that R1 becomes reserved by all lanes—odd lanes due to pending writes of $I_1$ and even lanes due to pending writes of $I_2$. At ⑥, $I_2$ writes the load value to R1 and hence it clears the even bits in the R-mask of register R1 in the scoreboard unit. Currently, R1 has an R-mask=0101 and it does not have any common active threads with $I_3$ which has an I-mask=1010. Hence, the dependency bit that corresponds to R1 in the dependency mask of $I_3$ is cleared and $I_3$ becomes eligible for scheduling.

### 3.3.3 Interaction with Barriers:

MP IPDOM enables additional functionality for barriers compared to the SPS model. In SPS, the entire divergent warp is suspended if any of its diverged threads encounters a barrier [3]. This is essentially because a stack-based execution model requires the top of the stack split to reach the reconvergence point before starting the execution of another split. If the top of the stack split is waiting at a barrier, the entire warp is suspended. Therefore, it is not possible to synchronize divergent threads of the same warp. In contrast, MP independently schedules diverged warp splits until they reach their reconvergence point enabling threads to synchronize within the same warp while diverged.

## 4 Opportunistic Early Reconvergence

In Section 2.2, we explained that the IPDOM reconvergence point is the earliest *guaranteed* reconvergence point. However, in certain situations, there are opportunities to reconverge at earlier points than the IPDOM point. Such situations depend on the outcomes of a sequence of branch instructions and the scheduling order of warp splits. Hence, early reconvergence is not guaranteed for all executions, but rather occurs dynamically when certain control paths are followed. Early reconvergence opportunities are common in applications with unstructured control flow [10, 8].

**Code Example:**

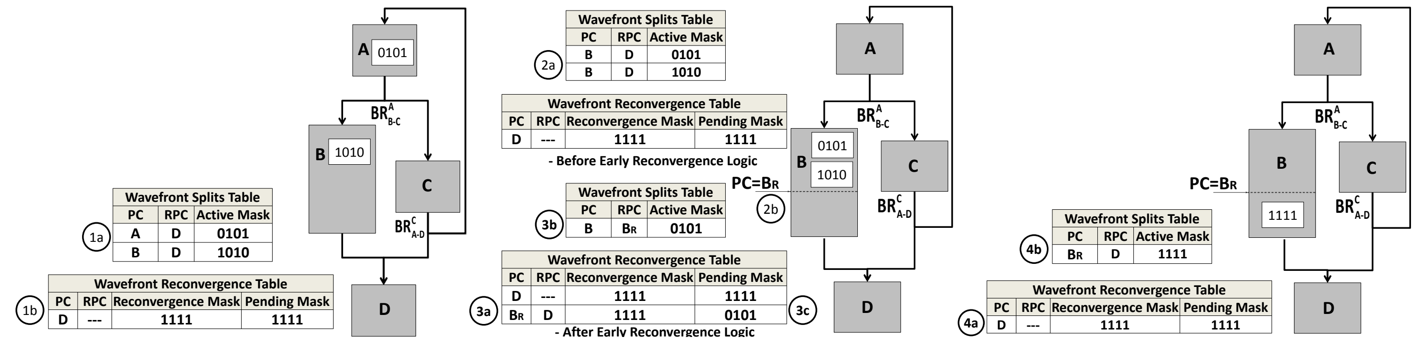
```
I0 : LOAD R0,  0(R5);
     if(id%2==0)
I1 :     LOAD R1,0(R0);
     else{
I2 :     LOAD R1,0(R4);
I3 :     ADD  R4,R0,R1;
     }
```

|  | Scoreboard |  | I-Buffer |  |  |
|---|---|---|---|---|---|
|  | Reg | R-mask | Inst. | Dep-mask | I-mask |
|  | - | - | $I_0$ | 00 | 1111 |
|  | - | - | - | - | - |
| ① | R0 | 1111 | $I_1$ | 01 | 0101 |
|  | - | - | $I_2$ | 00 | 1010 |
| ② | R0 | 1111 | $I_1$ | 01 | 0101 |
|  | R1 | 1010 | - | - | - |
| ③ | R0 | 1111 | $I_1$ | 01 | 0101 |
|  | R1 | 1010 | $I_3$ | 11 | 1010 |
| ④ | - | - | $I_1$ | 00 | 0101 |
|  | R1 | 1010 | $I_3$ | 10 | 1010 |
| ⑤ | - | - | - | - | - |
|  | R1 | 1111 | $I_3$ | 10 | 1010 |
| ⑥ | - | - | - | - | - |
|  | R1 | 0101 | $I_3$ | 00 | 1010 |

**Figure 9. MP IPDOM scoreboard example**

Figure 10 shows a code snippet that has unstructured control flow. We will use this code snippet to illustrate the modified operation of MP with support for opportunistic early reconvergence. Figure 11 shows instantaneous snapshots for a warp with four threads traversing through the control flow graph corresponding to the code in Figure 10. Active masks within basic blocks represent threads that are executing basic blocks at a specific time. Basic blocks with no active masks are not executed by any threads at that time. Figure 11a shows a snapshot of the execution where there are two warp splits, $A_{0101}$ and $B_{1010}$, executing basic blocks A and B respectively (1a). Both diverged warp splits have their IPDOM reconvergence point at D (1b). This initial state results if a divergence at $BR^{A}_{B-C}$ results in two splits $B_{1010}$ and $C_{0101}$, and split $C_{0101}$ reaches $BR^{C}_{A-D}$ before split $B_{1010}$ finishes executing basic block B.

Next, warp split $A_{0101}$ branches to basic block B after executing $BR^{A}_{B-C}$. This scenario creates an early reconvergence opportunity, where there are two splits ($B_{0101}$ and $B_{1010}$) of the same warp executing the same basic block B (2a). To detect an early reconvergence opportunity, an associative search within the ST using the first PC of the basic block is performed upon the insertion of any new entry. If there is an existing entry that matches the new entry in both the PC and RPC entries then an early reconvergence op-



Wavefront Splits Table (1a)

| PC | RPC | Active Mask |
|---|---|---|
| A | D | 0101 |
| B | D | 1010 |

Wavefront Reconvergence Table (1b)

| PC | RPC | Reconvergence Mask | Pending Mask |
|---|---|---|---|
| D | --- | 1111 | 1111 |

Wavefront Splits Table (2a)

| PC | RPC | Active Mask |
|---|---|---|
| B | D | 0101 |
| B | D | 1010 |

Wavefront Reconvergence Table

| PC | RPC | Reconvergence Mask | Pending Mask |
|---|---|---|---|
| D | --- | 1111 | 1111 |

Wavefront Splits Table (3b)

| PC | RPC | Active Mask |
|---|---|---|
| B | $B_R$ | 0101 |

Wavefront Reconvergence Table (3a, 3c)

| PC | RPC | Reconvergence Mask | Pending Mask |
|---|---|---|---|
| D | --- | 1111 | 1111 |
| $B_R$ | D | 1111 | 0101 |

Wavefront Splits Table (4b)

| PC | RPC | Active Mask |
|---|---|---|
| $B_R$ | D | 1111 |

Wavefront Reconvergence Table (4a)

| PC | RPC | Reconvergence Mask | Pending Mask |
|---|---|---|---|
| D | --- | 1111 | 1111 |

(a) Initial State (b) $A_{0101}$ executes $BR^{A}_{B\text{-}C}$ (c) $B_{1010}$ reconverges at an early reconvergence point

**Figure 11. Operation of the Multi-Path with the Opportunistic Reconvergence (OREC) enabled**

```
do{
    //BB_A
    if(cond1){
        //BB_B
        break;
    }else{
        //BB_C
    }
}while(cond2);
//BB_D
```

**Figure 10. Unstructured control flow**

portunity is detected. The early reconvergence point is the program counter of the next instruction of the leading warp split. In this example, $B_{1010}$ is the leading split, and the early reconvergence point is labeled $B_R$ (2b). A new entry is added to the RT to represent the early reconvergence point (3a). The RPC of the new entry ($B_{0101}$) is set to the early reconvergence point ($B_R$). Finally, warp split $B_{1010}$ in the ST is invalidated (3b), and accordingly the pending mask of the early reconvergence entry is updated (3c) (warp split $B_{1010}$ is already at the early reconvergence point).

The execution continues as explained in Section 3. Warp split $B_{0101}$ reaches the early reconvergence point $B_R$. Its entry in the ST is invalidated, the pending mask of the reconvergence entry $B_{R1111}$ is updated, and the reconvergence entry $B_{R1111}$ moves from the RT (4a) to the ST (4b). Similar cases to this example occur with more complex code in several GPU applications [8]. We evaluate the benefits of opportunistic reconvergence in Section 6.

## 5 Methodology and Benchmarks

We model MP IPDOM as described in Section 3 in GPGPU-Sim (version 3.1.0) [2]. Our modified GPGPU-Sim is configured to model a Geforce GTX 480 (Fermi) GPU with the configuration parameters distributed with GPGPU-Sim 3.1.0 but with 16KB instruction cache per core (see Section 6.4 for details). MP IPDOM does not restrict the scheduling order, so we can use any scheduler.

| Interleavable Benchmarks | | | |
|---|---|---|---|
| **Name** | **Abbr.** | **Name** | **Abbr.** |
| Monte Carlo | MC [1] | Mandelbrot | MAND [19] |
| 3-D Renderer | REND [27] | MUMMER | MUM[2] |
| MUMMER++ | MUMpp [11] | Memcached | MEMC [12] |
| Laplace Solver | LPS [2] | Ray Tracing | RAY [2] |
| LU Decomposition | | LUD [2] | |

**Table 1. Studied benchmarks**

For both the baseline (i.e., SPS model) and our MP variations, we use a greedy-then-oldest (GTO). GTO runs a single warp until it stalls then picks the oldest ready warp [22]. On the baseline, the GTO scheduler outperforms the Two-Level and the Loose Round Robin schedulers. For splits within the same warp, we use a simple Round Robin scheme to alternate between them. We model, the opportunistic reconvergence optimization described in Section 4 as a separate configuration.

In Section 6, we present results for MP IPDOM when it limits the number of concurrently supported warp splits. This is modelled by setting a maximum constraint on the active number of entries in the ST. If, upon a branch, a new entry is required to be inserted to the ST, and the table is already at its maximum capacity, the new warp split is not considered for scheduling until an ST entry is invalidated. The configuration with two entries models the effect of the Dual-Path Stack (DPS) with the *Path-Forwarding* optimization [21] (more details are provided in Section 7).

We study benchmarks from Rodinia [5] and those distributed with GPGPU-Sim [2]. We also use some benchmarks with multi-path divergence from other sources:

***MEMC:*** Memcached is a key-value store and retrieval system. It is described in detail by Hetherington et al. [12].

***REND:*** Renderer performs 3D real-time raytracing of triangle meshes. For benchmarking, we use pre-recorded frames provided with the benchmark [27].

***MC:*** MC-GPU is a GPU-accelerated X-ray transport simulation code that can generate clinically-realistic radiographic projection images and computed tomography scans of the human anatomy [1].

***MAND:*** Mandelbrot computes a visualization of mandelbrot set in a complex cartesian space. It partitions the space into pixels and assigns several pixels to each thread [19].

Note that out of 32 benchmarks, we only report the detailed results for the 9 benchmarks shown in Table 1, because the other benchmarks execute identically over SPS, DPS and MP variations. They perform identically because they are either non-divergent or they are divergent but all branches are one-sided branch hammocks such that the branch target is the reconvergence point. Under IPDOM reconvergence constraints, these applications do not exhibit parallelism between their basic blocks (i.e., *non-interleavable* [21]).

## 6 Experimental Results

This section presents our evaluation for the MP model.

### 6.1 SIMD Unit Utilization

Figure 12 shows the SIMD unit utilization ratio for benchmarks in Table 1. As expected, the SPS, DPS and basic MP have the same SIMD unit utilization because they all reconverge at the IPDOM reconvergence points. However, the opportunistic reconvergence optimization provides an average of 48% and up to 182% improvement in SIMD unit utilization. Benchmarks that exhibit unstructured control flow benefit from the opportunistic reconvergence.

### 6.2 Thread Level Parallelism

Figure 13 shows the average ratio of warp splits to warps. A value grater than one means an increase in the schedulable entities at the scheduler. Hence, it is more likely for the scheduler to find an eligible split to schedule. The SPS exposes only one split at a time to the scheduler. Hence, its average ratio of warp splits to warps is always one.

As shown in the figure, benchmarks such as ***REND***, ***MAND***, ***MC***, ***MUM*** and ***MEMC*** show a considerable increase in the average ratio of the warp splits as the maximum number of supported warp splits increase. The MP IPDOM achieves ∼50%-690% increase in the warp splits compared to SPS, and ∼11%-400% compared to the DPS. This is mainly because some of these benchmarks have highly unstructured control flow (e.g. ***REND***, ***MAND*** and ***MC***) and they also have *switch* and *else if* statements that increase the number of parallel control flow paths (e.g. ***MEMC***, ***MUM*** and ***MC***).

The ***MUMpp***, ***LPS***, ***LUD*** and ***RAY*** benchmarks have a limited increase in the average number of warp splits (∼-5%). This is mainly for two reasons. The ***LPS***, ***LUD*** and ***RAY*** benchmarks have SIMD utilization (>75%), hence, for a large portion of time there is no divergence at all. This biases the average towards a single split per warp. Also, the four benchmarks have a maximum of two parallel control flow paths at a time, otherwise, they are dominated by single-sided branches (i.e., one of their two outcomes is the branch reconvergence point). Therefore, for these applications MP acts identically to DPS.

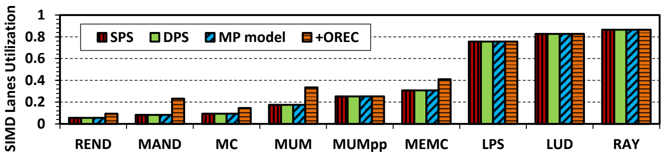


**Figure 12. SIMD units utilization**

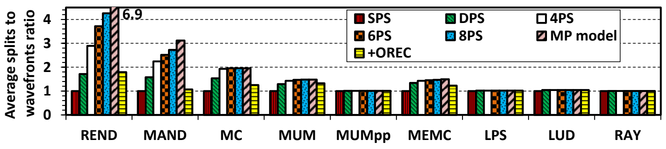


**Figure 13. Warp splits to warp ratio**

The data in Figure 13 shows that the opportunistic early reconvergence bounds the increase in the number of warp splits. This is expected because it forces the splits to wait for each other at the early reconvergence points, and it tends to combine multiple splits into a single one.

Figure 14 shows the average breakdown of the threads' state at the scheduler. The threads' state means whether the thread can issue its next instruction (i.e., *eligible*) or not. Also, it breaks down the different possible reasons that stalls a thread. Since a single thread can be stalled due to more than one reason (e.g., data hazard and structural hazard at the same time), the breakdown assumes priorities for the different possible stalling reasons. The priority order is the order in Figure 14 from bottom to top.

“Suspended" threads are those stalled due to divergence (i.e., they are either waiting at reconvergence points or they are waiting at a parallel control flow path). There is a gradual decrease in the number of “Suspended" threads for mechanisms that support a larger number of warp splits. For example, on ***MAND***, MP has an average number of suspended threads that is ∼35% less than SPS.

However, the decrease in the average number of suspended threads does not directly translate to an increase in eligible threads. In particular, the non-suspended threads can be stalled due to data or structural hazard. This effect is pronounced in the ***MUM*** and ***MEMC*** benchmarks, where the gradual decrease in the average suspended threads is followed by a gradual increase in the average threads stalled due to structural hazard.

The ***REND*** benchmarks suffers from load imbalance, where some warps exit the kernel early while others continue execution under divergence. The splits of the diverged warps are serialized in SPS. This biases the average result to have a large portion of "Finished" threads on the SPS model. As we increase the number of allowed warp splits, the scheduler interleaves the execution of diverged splits

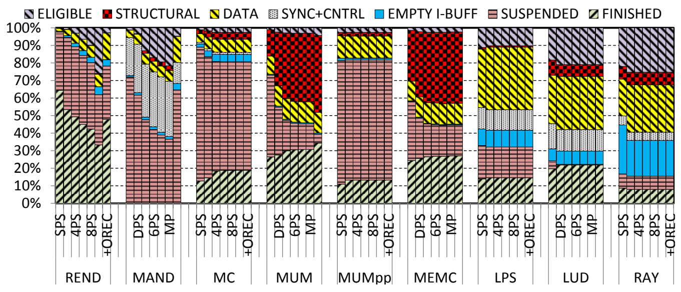


**Figure 14. Average breakdown of threads' state at the scheduler**

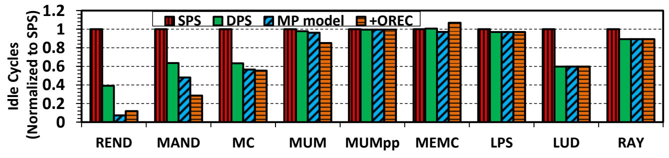


**Figure 15. Idle cycles**

which in turn speeds up their execution. Hence, on average, we have more "FINISHED" threads.

## 6.3 Idle Cycles

Figure 15 shows the idle cycles accumulated for all cores for our benchmarks. The increase in the average number of eligible threads in Figure 14 translates into a decrease in the Idle cycles. Only the ***MEMC*** benchmark shows ∼7% increase in the idle cycles when we adapt MP with opportunistic reconvergence compared to SPS model. We discuss this in detail in Section 6.4.

## 6.4 Impact on Memory System

*Instruction Locality:* MP has a direct impact on instruction cache locality. Unlike the SPS model, MP may require frequent fetching of instructions from distant blocks in the code. While this is not a problem for most GPU kernels which tend to have small static code size, it can considerably affect the instruction cache misses in a large kernel. We find that the effect of instruction misses on the overall performance is negligible with a 16 KB cache. Figure 16 shows the instruction cache misses normalized to the SPS model. There is a considerable increase in instruction cache misses for the ***REND*** benchmark but it has limited effect on the overall performance (only an average of up to 2.5% of threads are stalled due to empty instruction buffers; see Figure 14). Since instruction fetch is done at warp splits granularity, MP with opportunistic reconvergence tends to have less instruction cache accesses and misses.

*Data Locality:* The effect of MP execution on data cache locality depends on the application and whether parallel control flow paths access contiguous memory locations or not. Figure 17 shows the L1 data cache misses normalized to the SPS execution model. The ***MUM*** benchmark has reduced L1 data cache misses in MP compared to the

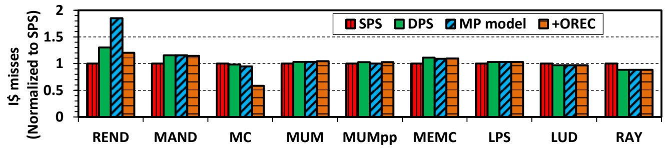


**Figure 16. Inst. cache misses (16KB I$)**

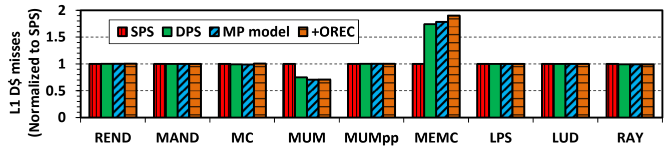


**Figure 17. L1 data cache misses (32KB D$)**

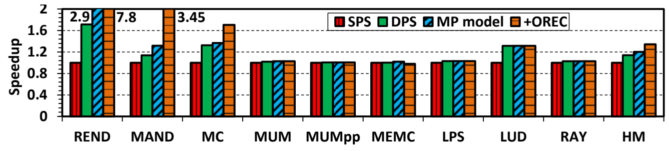


**Figure 18. Overall speedup**

SPS model, but it does not have a big impact on its performance because it already has low data cache misses (<0.3 MPKI "misses per thousand instructions"). However, the MEMC benchmark suffers from a significant increase in its misses (> 80%). In particular, the total misses jumps from 30 MPKI to 82 MPKI. In depth analysis suggests that the MEMC benchmark loses its intra-warp locality. That is, warp splits evict each others' data from the data cache before they get accessed again by the same warp split. These observations are consistent with prior work [22].

## 6.5 Overall Performance

Figure 18 shows the speedup over SPS. The speedup comes mainly due to the reduced idle cycles (i.e., more warp split instructions per cycle) and the improved SIMD units' utilization (i.e., more throughput per warp split instruction). MP with opportunistic reconvergece has 32% harmonic mean speedup over the SPS model, compared to 18.6% and 12.5% for the basic MP and DPS models.

## 6.6 Implementation Complexity

As discussed in Section 3.3, implementing MP requires modifications to the branch unit and the scoreboard logic. First, we replace the stack with two tables (ST and RT). Figure 19 shows the maximum usage of both tables. Although the maximum usage of entries in the ST and RT can reach large numbers with the basic MP, the opportunistic reconvergence helps to bound this increase to a maximum of 12 and 13 entries respectively.

The modified scoreboard logic adds 1.5KB storage requirement for 48 warps per SM each with 8 register entries (GTX480 configuration). We synthesized both the basic scoreboard and the modified scoreboard on NCSU

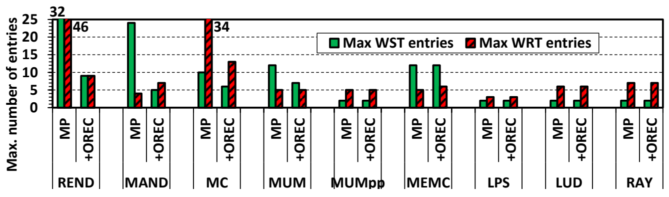


**Figure 19. Usage of the WST and RT tables**

FreePDK 45nm [24]. We model the scoreboard as a small set associative SRAM. The synthesis results estimates an area 175,432 $\mu m^2$ and a total power of $\sim 4.4 mW$ total power at 50% activity factor; compared to 91,365 $\mu$m$^2$ and $1 mW$ power for the original scoreboard. The SRAM used is based on NCSU's FabScalar memory compiler, FabMem [23], with 6 read ports and 3 write ports.

We also use GPU-Wattch [15] to estimate the increase in the dynamic power due to the associated with overall increased performance. For all our benchmarks, we find that the maximum observed increase in the average dynamic power is (37.5%) for the REND benchmark. However, the $7\times$ speedup justifies such increase in power.

## 7 Related Work

Dual Path Stack (DPS) is a recent proposal that extends the SPS stack to support two concurrent paths of execution [21] while maintaining reconvergence at immediate postdominators. Instead of stacking the taken and not-taken paths one after the other, the two paths are maintained in parallel. DPS maintains separate scoreboard units for each path to avoid false dependencies between independent splits. However, it is necessary to check both units to make sure there are no pending dependencies accorss divergence and reconvergence points. As shown in Section 6, DPS has limited benefits on benchmarks that have multi-path divergence or benchmarks that have unstructured control flow behavior. The DPS also suffers from the same limitations related to interaction with barriers as the SPS model.

Dynamic Warp Subdivision (DWS) adds a warp splits table to the conventional stack [17]. Upon a divergent branch, it uses heuristics to decide which branches start subdividing a warp into splits and which do not. If a branch subdivides a warp, DWS ignores IPDOMs nested in that branch. This often degrades DWS performance compared to the SPS model [21]. Unlike DWS, MP IPDOM manages to maximize TLP under the IPDOM reconvergence constraints.

Similar to DPS, Simultaneous Branch Interweaving (SBI) allows a maximum of two warp splits to be interleaved [4]. However, SBI targets improving SIMD utilization by spatially interleaving the diverged warp splits on the SIMD lanes. The reconvergence tracking mechanism proposed with the SBI requires constraints on both the code layout and the warp splits' scheduling priorities to adhere to thread-frontier based reconvergence [8].

Dynamic Warp Formation (DWF) is not restricted to IPDOM reconvergence [9]. Instead, it opportunistically group threads that arrive at the same PC, even though they belong to different warps. DWF performance is highly dependent on the scheduling to increase the opportunity of forming denser warps, and sometimes leads to starvation eddies.

Temporal SIMT (T-SIMT) is a new microarchitecture where each warp is mapped to a single lane, and the threads within a warp dispatch an instruction one after the other over successive cycles [14]. Upon divergence, threads progress independently; and hence divergence does not reduce the SIMD units utilization. However, reconvergence is still favourable to perform memory address coalescing and scalar operations [14]. The T-SIMT microarchitecture lacks a hardware mechanism to track reconvergence of diverged warp splits, therefore, they insert (syncwarp) instructions at the immediate postdominator of the top-level divergent branches [14]. Our MP microarchitecture provides a hardware mechanism to track nested reconvergence points. Hence, it can be integrated with T-SIMT.

Multiple SIMD Multiple Data (MSMD) [28] proposes quite large changes to the baseline architecture to support flexible SIMD datapaths that can be repartitioned among multiple control flow paths. Similar to T-SIMT, MSMD proposes to use a special syncronization instruction to reconverge at postdominators, however, the paper does not specify an algorithm that determines where to place these syncronization instructions and how to determine which specific threads to synchronize at each instruction.

Thread Block Compaction (TBC) and TBC-like techniques allow a group of warps to share the same SIMT stack [10, 18]. Hence, at a divergent branch, threads from grouped warps are compacted into new more dense warps. Since TBC employs a thread block wide stack, it suffers more from the reduced thread level parallelism [20]. This makes MP IPDOM a good candidate to integrate with TBC to mitigate such deficiencies. For this purpose, the warp-wide divergence and reconvergence tables would need to be replaced with thread block wide tables.

## 8 Conclusion

In this paper, we propose a novel mechanism which enables efficient multi-path execution in GPUs. In particular, our mechanism enables tracking IPDOM reconvergence points of diverged warp splits while interleaving their execution. This is achieved by replacing the stack-based structure that handles both the divergence and reconvergence in the current GPUs with two tables. One table tracks the concurrent executable paths upon every branch, while the other tracks the reconvergence points of these branches. Furthermore, we illustrate that our multi-path model can be modified to enable opportunistic early reconvergence at run-time to improve SIMD units utilization for applications with unstructured control flow behavior. Evaluated on a set of benchmarks with multi-path divergent control flow, our proposal achieves 32% speedup over conventional single-path SIMT execution.

## Acknowledgments

The authors would like to thank Wilson Fung, Tayler Hetherington, Ali Bakhoda, Timothy G. Rogers, Ayub Gubran, Hadi Jooybar and the reviewers for their insightful feedback. This research was funded in part by a Four Year Doctoral Fellowship from University of British Columbia, the Natural Sciences and Engineering Research Council of Canada and a grant from Advanced Micro Devices Inc.

## References

[1] A. Badal and A. Badano. Accelerating Monte Carlo Simulations of Photon Transport in a Voxelized Geometry Using a Massively Parallel Graphics Processing Unit. *Medical physics*, 36:4878, 2009.

[2] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *Proc. IEEE Symp. on Perf. Analysis of Systems and Software (ISPASS)*, pages 163–174, 2009.

[3] A. Betts, N. Chong, A. Donaldson, S. Qadeer, and P. Thomson. GPUVerify: a Verifier for GPU Kernels. In *Proc. ACM Int'l Conf. on Object oriented programming systems languages and applications*, pages 113–132, 2012.

[4] N. Brunie, S. Collange, and G. Diamos. Simultaneous Branch and Warp Interweaving for Sustained GPU Performance. In *Proc. IEEE/ACM Symp. on Computer Architecture (ISCA)*, pages 49–60, 2012.

[5] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proc. IEEE Symp. on Workload Characterization (IISWC)*, pages 44–54, 2009.

[6] S. Collange. Stack-less SIMT Reconvergence at Low Cost. Technical Report hal-00622654, ARENAIRE - Inria Grenoble Rhône-Alpes / LIP Laboratoire de l'Informatique du Parallélisme, 2011.

[7] B. W. Coon, P. C. Mills, S. F. Oberman, and M. Y. Siu. Tracking Register Usage during Multithreaded Processing Using a Scoreboard having Separate Memory Regions and Storing Sequential Register Size Indicators. US Patent 7,434,032, 2008.

[8] G. Diamos, B. Ashbaugh, S. Maiyuran, A. Kerr, H. Wu, and S. Yalamanchili. SIMD Re-convergence at Thread Frontiers. In *Proc. IEEE/ACM Symp. on Microarch. (MICRO)*, pages 477–488, 2011.

[9] W. Fung, I. Sham, G. Yuan, and T. Aamodt. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *Proc. IEEE/ACM Symp. on Microarch. (MICRO)*, pages 407–420, 2007.

[10] W. W. L. Fung and T. M. Aamodt. Thread Block Compaction for Efficient SIMT Control Flow. In *Proc. IEEE Symp. on High-Perf. Computer Architecture (HPCA)*, pages 25–36, 2011.

[11] A. Gharaibeh and M. Ripeanu. Size Matters: Space/Time Tradeoffs to Improve GPGPU Applications Performance. In *Proc. ACM Int'l Conf. for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–12, 2010.

[12] T. H. Hetherington, T. G. Rogers, L. Hsu, M. O'Connor, and T. M. Aamodt. Characterizing and Evaluating a Key-Value Store Application on Heterogeneous CPU-GPU Systems. In *Proc. IEEE Symp. on Perf. Analysis of Systems and Software (ISPASS)*, pages 88–98, 2012.

[13] S. Keckler, W. Dally, B. Khailany, M. Garland, and D. Glasco. GPUs and the Future of Parallel Computing. *Micro, IEEE*, 31(5):7–17, 2011.

[14] Y. Lee, R. Krashinsky, V. Grover, S. Keckler, and K. Asanovic. Convergence and Scalarization for Data-Parallel Architectures. In *Proc. IEEE/ACM Symp. on Code Generation and Optimization (CGO)*, pages 1–11, 2013.

[15] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi. GPUWattch: Enabling Energy Optimizations in GPGPUs. In *Proc. IEEE/ACM Symp. on Computer Architecture (ISCA)*, 2013.

[16] A. Levinthal and T. Porter. Chap — A SIMD graphics processor. *SIGGRAPH*, 18(3):77–82, 1984.

[17] J. Meng, D. Tarjan, and K. Skadron. Dynamic Warp Subdivision for Integrated Branch and Memory Divergence Tolerance. In *Proc. IEEE/ACM Symp. on Computer Architecture (ISCA)*, pages 235–246, 2010.

[18] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt. Improving GPU Performance via Large Warps and Two-Level Warp Scheduling. In *Proc. IEEE/ACM Symp. on Microarch. (MICRO)*, pages 308–317, 2011.

[19] NVIDIA. CUDA SDK 3.2, September 2013.

[20] M. Rhu and M. Erez. CAPRI: Prediction of Compaction-Adequacy for Handling Control-Divergence in GPGPU Architectures. In *Proc. IEEE/ACM Symp. on Computer Architecture (ISCA)*, pages 61–71, 2012.

[21] M. Rhu and M. Erez. The Dual-Path Execution Model for Efficient GPU Control Flow. In *Proc. IEEE Symp. on High-Perf. Computer Architecture (HPCA)*, pages 235–246, 2013.

[22] T. G. Rogers, M. O'Connor, and T. M. Aamodt. Cache-Conscious Wavefront Scheduling. In *Proc. IEEE/ACM Symp. on Microarch. (MICRO)*, pages 72–83, 2012.

[23] T. Shah. FabMem: A Multiported RAM and CAM Compiler for Superscalar Design Space Exploration. Master's thesis, North Carolina State University, 2010.

[24] J. Stine, I. Castellanos, M. Wood, J. Henson, F. Love, W. Davis, P. Franzon, M. Bucher, S. Basavarajaiah, J. Oh, and R. Jenkal. FreePDK: An Open-Source Variation-Aware Design Kit. In *Proc. IEEE of Microelectronic Systems Education (MSE)*, pages 173–174, 2007.

[25] AMD Corporation. *R700-Family Instruction Set Architecture*. 2011.

[26] Intel Corporation. *Intel HD Graphics OpenSource Programmer Reference Manual*. 2010.

[27] T. Tsiodras. Real-time raytracing: Renderer. http://users.softlab.ntua.gr/∼ttsiod/renderer.html, September 2013.

[28] Y. Wang, S. Chen, J. Wan, J. Meng, K. Zhang, W. Liu, and X. Ning. A Multiple SIMD, Multiple Data (MSMD) Architecture: Parallel Execution of Dynamic and Static SIMD fragments. In *Proc. IEEE Symp. on High-Perf. Computer Architecture (HPCA)*, pages 603–614, 2013.