Maximizing SIMD Resource Utilization in GPGPUs with SIMD Lane Permutation

Minsoo Rhu
Electrical and Computer Engineering Dept.
The University of Texas at Austin
minsoo.rhu@utexas.edu

Mattan Erez
Electrical and Computer Engineering Dept.
The University of Texas at Austin
mattan.erez@mail.utexas.edu

ABSTRACT

Current GPUs maintain high programmability by abstracting the SIMD nature of the hardware as independent concurrent threads of control with hardware responsible for generating predicate masks to utilize the SIMD hardware for different flows of control. This dynamic masking leads to poor utilization of SIMD resources when the control of different threads in the same SIMD group diverges. Prior research suggests that SIMD groups be formed dynamically by compacting a large number of threads into groups, mitigating the impact of divergence. To maintain hardware efficiency, however, the alignment of a thread to a SIMD lane is fixed, limiting the potential for compaction. We observe that control frequently diverges in a manner that prevents compaction because of the way in which the fixed alignment of threads to lanes is done. This paper presents an in-depth analysis on the causes for ineffective compaction. An important observation is that in many cases, control diverges because of programmatic branches, which do not depend on input data. This behavior, when combined with the default mapping of threads to lanes, severely restricts compaction. We then propose SIMD lane permutation (SLP) as an optimization to expand the applicability of compaction in such cases of lane alignment. SLP seeks to rearrange how threads are mapped to lanes to allow even programmatic branches to be compacted effectively, improving SIMD utilization up to 34% accompanied by a maximum 25% performance boost.

Categories and Subject Descriptors

C.1.2 [**Multiple Data Stream Architectures**]: Single-instruction-stream multiple-data-stream processors (SIMD)

General Terms

Design, Experimentation

Keywords

GPU, SIMD, SIMT, Control Divergence

1. INTRODUCTION

Recent GPUs adopt the *single-instruction multiple-thread* (SIMT) execution model, which can use efficient hardware organizations while presenting a simple parallel abstraction to the programmer. With SIMT, GPUs can utilize efficient *single-instruction multiple-data* (SIMD) pipelines while still supporting arbitrary control flow in each thread. GPUs have native hardware support for conditional branches, allowing each (SIMD) lane to execute its own logical thread. Despite the hardware support, mapping control-flow intensive applications to GPUs is challenging. When a conditional branch is evaluated such that a subset of the SIMD lanes execute the taken path and others the non-taken path, the underlying SIMD hardware serializes the two paths and each path only occupies a subset of the SIMD lanes.

Recent work [14, 15, 25] has demonstrated that the performance degradation caused by this *control divergence* can be alleviated by dynamically re-grouping SIMD instructions from a larger collection of threads (Section 2.3). The collections of threads are referred to as *cooperating thread arrays* (CTAs) or *thread blocks* by NVIDIA's CUDA [28] and *work-groups* by OpenCL [4, 5]; and the SIMD instructions are known as *warps* in CUDA and *wavefronts* in OpenCL. The idea is that the large number of warps within a CTA offers opportunities to *compact* execution and fill in idle lanes in one warp with active threads from another warp [14, 25, 31].

We address a significant issue with previously proposed compaction mechanisms that limits their applicability. To simplify the structure of the register-file, current GPUs statically assign a fixed SIMD lane to each thread based on its thread-ID in a sequential manner. As we show in this paper, however, such a fixed mapping limits the potential *compactability* of branches. A branch is *compactable* if compaction can reduce the number of idle lanes in either the true or the false path. We refer to some compactable branches as *aligned*; when threads that diverged into the same path are clustered (aligned) on a common subset of SIMD lanes. Such threads cannot be compacted effectively because the active and inactive threads are aligned between warps. We observe that in many cases, *aligned divergence* originates from a branch whose condition is dependent only on programmatic values (e.g., CTA-/warp-/thread-ID, scalar input parameters), in which case the divergence pattern exhibits similarly across warps. We present the first in-depth quantitative analysis on the cause of aligned divergence.

Based on the insights of the analysis, we propose and evaluate *SIMD lane permutation* (SLP) which improves the compaction behavior of divergent branches. SLP permutes

the home SIMD lanes of threads such that threads with a similar control flow do not execute in a common SIMD lane. Using SLP, diverging paths that are non-compactable as-is can potentially be transformed into compactable ones. This expands the applicability of compaction thus leading to better performance. While the idea of applying a permutation to alleviate scheduling conflicts is not new (e.g., [9, 15, 22, 23, 30, 41]), we make the following contributions in the context of thread compaction:

- We identify previous compaction mechanisms as being limited in effectiveness by aligned divergence. Using a GPU emulator, we provide a quantitative and qualitative analysis on the root cause of aligned divergence.
- We introduce the concept of SLP that enables higher compactability of branches. By revisiting some preliminary studies on permutation, we quantitatively show why one works better than the other. Based on this, we carefully design a new *balanced* permutation, that is highly robust and well-suited to the branching behavior we observe in CUDA applications.
- We evaluate the benefits of SLP in terms of compactability, SIMD lane utilization, and performance using a set of CUDA workloads; SLP provides up to a 34% increase in SIMD lane utilization accompanied by a maximum 25% performance improvement.

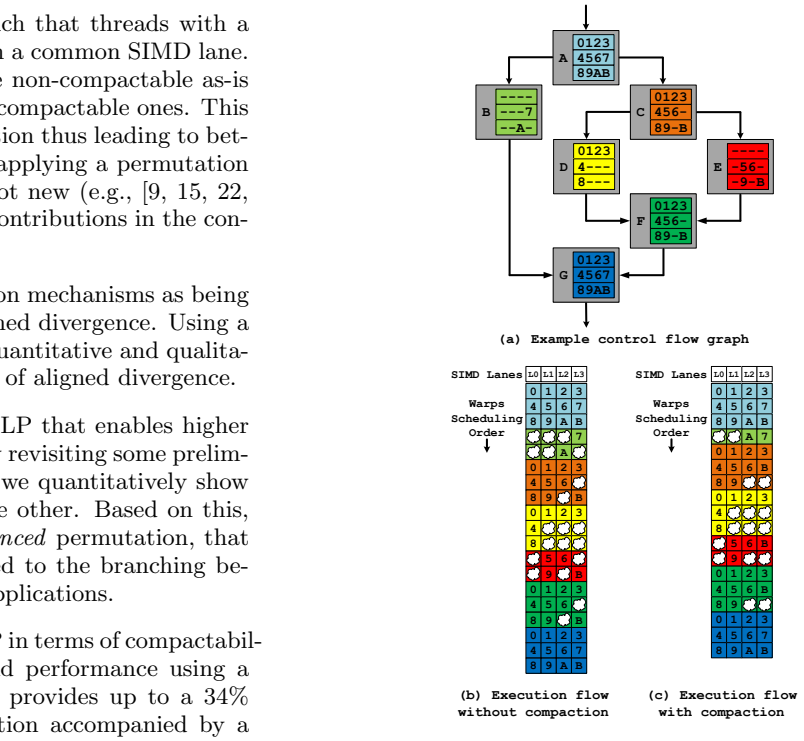


Figure 1: Example showing how a control flow graph (a) is executed without compaction (b) and with compaction (c). Each CTA contains 3 warps of 4 threads each. The numbers represent the thread-IDs that are executing in a basic block, while "–" denotes masked-out threads.

2. BACKGROUND AND RELATED WORK

2.1 CUDA Programming Model

Current GPUs, such as NVIDIA's Fermi [26], consist of multiple shader cores (streaming multiprocessors (SM) in NVIDIA terminology), where each SM contains a number of parallel execution lanes (referred to as CUDA cores by NVIDIA) that operate in SIMD fashion: a single instruction is issued to each of the parallel SIMD lanes and that instruction is repeated multiple times per lane. In CUDA, a single program (the *kernel*) is executed by all the threads that are spawned for execution. The programmer groups the threads into *CTA*s, where each CTA consists of multiple *warps*. In Fermi, each SM schedules the instruction to be executed in a warp of 32 *threads*, yet the SIMD grouping of warps is not explicitly exposed to the CUDA programmer [28]. The number of threads that constitute a CTA is an application parameter, and a CTA typically consists of enough threads to form multiple warps. In the rest of this paper, we will mainly use the terms defined in CUDA [28].

2.2 Control Divergence

Although GPU hardware is modeled for SIMD execution, the execution model enables each thread to maintain its own logical control flow. The hardware generated active bitmasks (predicates), designate whether a thread is active or not, allowing independent branching for each thread. Because threads that are masked out do not commit the results of their computation, only threads that are active actually execute the instruction; thus enabling SIMT, however, partially serializes execution on a divergent branch as the true and false path must be executed one after another with those threads on the non-active path being masked out (shown as "bubbles" in some SIMD lanes in Figure 1). Accordingly, each divergence reduces SIMD lane utilization because more threads are masked out from execution.

To alleviate the resource underutilization, proposals were made to have threads from the true and false path reconverge at the immediate post-dominator [15, 37] (the first instruction of basic block F and G in Figure 1 (a)) in the control flow graph, rather than serializing execution until threads complete execution. Such reconvergence point of a branch is computed at compile time and is tracked by a hardware reconvergence stack. Current NVIDIA GPUs adopt such reconvergence stack structure per-warp, tracking the points of divergence and reconvergence of threads within a warp. Each stack entry tracks the active mask associated with each control flow path (hence each basic block in a control flow graph), its program counter, and its reconvergence point. By always scheduling active threads at the top of its stack, only threads executing in a common path are issued thereby avoiding any structural hazard of the SIMT model. Figure 1 (b) provides a high-level overview of how warps are scheduled in executing the control flow graph of Figure 1 (a) without compaction (*No_TBC*). While the divergent branch at the end of block A and C reduces the SIMD lanes occupied, such inefficiency is minimized by reconverging the diverged paths at the immediate post-dominator (block F and G).

2.3 Compaction for Better SIMD Utilization

While the aforementioned mechanism correctly handles even nested divergent branches, each divergence reduces the number of active threads and increases the number of wasted execution slots. Several mechanisms have been proposed to mitigate the magnitude of waste, including compaction-

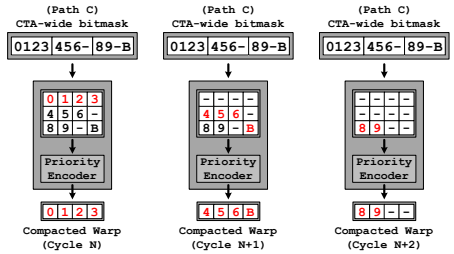


Figure 2: Example showing how the priority encoders compact the CTA-wide bitmask (at basic block C of Figure 1) into three warps ($NW_{TBC} = 3$). Although we explicitly use the thread-IDs to illustrate the bitmask, the actual hardware uses single bits. Note that threads with thread-ID 0, 4, and 8 always execute in the leftmost SIMD lane, as each thread's home SIMD lane is statically determined using their thread-IDs. Compaction is ineffective in the above example as NW_{TBC} equals NW_{NoTBC}.

based architectures [14, 25, 31]. The basic idea of improving utilization through compaction is to consider multiple warps, up to the entire CTA, as a single unit when a branch diverges and reconverges. Instead of allowing each warp to be serially executed, threads executing in a common basic block dynamically form new warps to minimize masked execution slots. *Thread block compaction* (TBC) [14], for instance, considers the entire CTA as a single unit when a branch diverges. TBC considers the active mask of the entire CTA and dynamically compacts it on all diverging paths (true/false paths) and reconvergence points. To maintain hardware efficiency, compaction is activated while maintaining the fixed association of each thread to its home SIMD lane. The two threads executing in Figure 1–path B (thread-ID = 7 and A), for instance, can be compacted into a single warp (thus having better SIMD lane utilization) as both threads are executing in different SIMD lanes. As illustrated in Figure 2, however, threads in path C are not compactable as thread-ID 0, 4, and 8 are all aligned in the leftmost lane, preventing compaction. Compaction is performed by a set of priority encoders (Figure 2) that leverage the CTA-wide mask to identify the minimum number of warps to execute the active threads. When the number of active warps generated through compaction (NW_{TBC}) is smaller than the number of warps needed to execute without compaction (NW_{NoTBC}), SIMD lane utilization is improved (e.g., path B in Figure 1 (c)).

One downside to TBC is that it can degrade performance because compaction requires that all threads in a CTA synchronize on all conditional branches. Rhu et al. [31] introduced the idea of predicting whether a diverging path is likely to be compactable, so that such synchronization overhead of compaction is reduced.

2.4 Related Work

The focus of this paper is on SIMD lane permutation (SLP) that increases the effectiveness of compaction. We summarize prior work that is closely related to the idea of permutation below and do not include a discussion of work related to compaction itself beyond the descriptions above. For more discussions on compaction hardware and general divergence issues in recent GPUs, we refer the interested readers to the discussions in [11, 14, 24, 25, 31, 32, 38, 39].

Using randomized permutation and hashing to mitigate scheduling conflicts on shared resources and improve load balance has been studied in many contexts in the past. Examples of techniques that are related to SLP include resource-allocation algorithms for networks and switches [6, 21, 35] and compute load balance [13, 29]. A more relevant example is the use of deterministic and randomized permutations to reduce memory bank conflicts [22, 23, 30, 41]. Zhang et al. [41], for example, used *XOR*-permuted bank indexing to reduce bank conflicts from cache-induced accesses, which is conceptually similar to SLP. While not targeting compaction-based architectures, Brunie et al. [9] discussed some lane shuffling mechanisms in the context of dual-instruction multiple-thread architectures. Their use is, in some ways, more directly about load balancing than our use for compaction, but there are similarities with SLP and we compare their best performing permutations (*WID* and *Rev_WID*) in Section 6. We also evaluate the only permutation mechanism previously mentioned in the context of compaction. Fung et al. [15] adopted a simple permutation (detailed in Section 4) to mitigate lane conflicts for compaction, but we demonstrate cases where this permutation falls short. Our work goes much beyond these related studies by analyzing the root cause of lane conflicts in compaction-based GPU pipelines, designing a permutation from first-principles that specifically targets these sources of inefficiency, and evaluating lane permutation in detail.

We briefly summarize related but less directly relevant work below. Cray-I [33] and Illiac-IV [8] were early vector machines that incorporated masked execution. Cray-I contained a vector mask for vectorizing if-else statements and the Illiac-IV had a mode bit for turning each processing module on or off. These are conceptually similar to the active masks used in GPUs, but are explicit in software. *Predication* [3] is another possible implementation of masked execution, which is amenable to compiler optimization. Smith et al. [36] proposed *density-time execution* for vector architectures. Density-time execution is similar to compaction but operates in time with traditional masked vector execution, rather than across lanes. Similar studies have been done by Kapasi et al. for stream processors [20].

Table 1: Evaluated CUDA benchmarks.

Abbreviation	Description	#Instr.	Ref.
BACKP	Back propagation	1M	[10]
LPS	3D laplace solver	985K	[17]
BFS	Breadth first search	256K	[19]
MUM	MUMmerGPU	2.7M	[34]
BITONIC	Bitonic sort	2K	[27]
REDUCT	Reduction (Kernel 0)	44K	[27]
MDBROT	Mandelbrot	8.3M	[27]
DXTC	DXT compression	955K	[27]
AOSSORT	AOS sorting	39K	[27]
FDTD3D	FDTD stencil on 3D	71M	[27]
SORTNW	Sorting network	2M	[27]
EIGENVL	Eigen-value	6.7M	[27]
3DFD	Finite diff. comp. 3D	29K	[27]
DWTHARR	Harr wavelets	2K	[27]
QSRDM	Quasirandom generator	3.1M	[27]
BINOM	Binomial options	176K	[27]
CONVSEP	Separable convolution	204K	[27]
SOBFLT	Sobel filter	575K	[27]

3. ANALYSIS OF SIMT COMPACTION

3.1 Current Compaction Limitations

Figure 3 shows the average SIMD lane utilization ($SIMD_{util}$) of several benchmarks (see Table 1), which exhibit branch

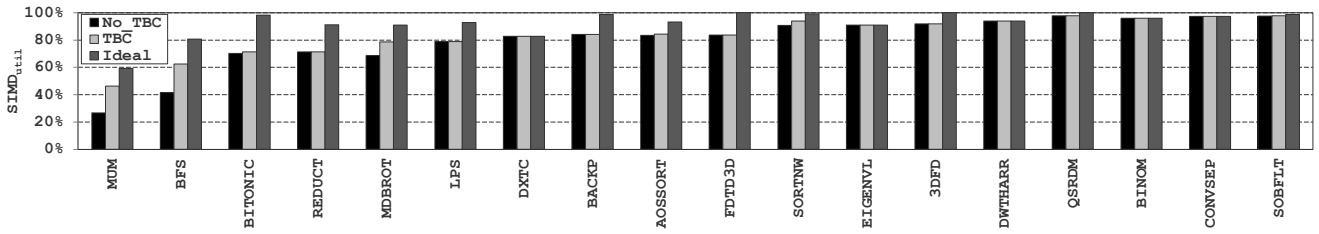


Figure 3: Average SIMD lane utilization of divergent benchmarks (Table 1), without compaction (No_TBC), with a current branch compaction technique (TBC), and with ideal compaction (TBC_{ideal}). SIMD lane utilization is defined as the fraction of SIMD lanes occupied (active) when a warp is executing. In order to isolate the effect of idle cycles, we only average the SIMD lane utilization of issued warps with at least a single thread active.

divergence, using each of three configurations: baseline without compaction, TBC (thread block compaction), and ideal compaction (TBC_{ideal}). With ideal compaction, threads can change SIMD lanes for optimal compaction. Changing SIMD lanes is not practical because of the required interconnect in the register file, but TBC_{ideal} provides an upper bound on any compaction within a CTA.

Overall, 11 of the 18 benchmarks show noticeable improvement in $SIMD_{util}$ with ideal compaction (another 3 have improvement of less than 2%). Interestingly, while all applications with low $SIMD_{util}$ can benefit from compaction, even some applications with relatively high $SIMD_{util}$ show significant potential (e.g., LPS, BACKP, AOSSORT, FDTD3D, SORTNW, and 3DFD). It is also important to note that TBC is unable to approach the full potential of compaction for any of the benchmarks. Furthermore, some of the high-$SIMD_{util}$ benchmarks actually benefit more from the current TBC implementation than some with low $SIMD_{util}$, for example, FDTD3D shows a 17% benefit, compared to just 13% exhibited by LPS.

While a benchmark's absolute $SIMD_{util}$ is certainly correlated with its compactability, it is not the sole factor that determines it. Rather, *how* the divergence manifests among warps is more critical because only threads that have diverged to the same path *and* do not share a common home SIMD lane can be compacted together (Figure 2). Previous compaction-mechanisms [14, 25, 31] are therefore only effective on divergent branches that do not cause active threads to align with a few common SIMD lanes (which we refer to as *aligned divergence* in the rest of this paper). For example, the three active threads in Figure 2 are fully aligned at the leftmost SIMD lane. As a result, compaction provides no benefit and NW_{TBC} and NW_{NoTBC} both equal 3. Such aligned branches are fairly common, as we discuss in the rest of this section.

3.2 Aligned Divergence

What fundamentally decides the control flow of each thread is how the branch predicate is evaluated. Aligned divergence is a phenomenon where threads with a common home SIMD lane have their predicate condition resolved the same way, causing active threads to be concentrated on a subset of the SIMD lanes. We observe that such alignment is rarely exhibited when the predicate depends on input data arrays. In such data-dependent branches (*D-branches*), different threads most likely reference different values thereby resolving the predicate differently (Figure 4 (a)). The divergence behavior of branches with a predicate condition that does not depend on a data array value (we refer to such non-data array values as *programmatic* values) is substantially different from the behavior of D-branches.

A *programmatic* value is viewed the same way across the threads. The indices for the CTA-ID (`blockIdx`), width and height of a CTA (`blockDim`), and scalar input parameters of a CUDA kernel (e.g., *imageW* in Figure 4 (d)), are all seen as constant values to all the threads within a CTA and are therefore programmatic values. In addition, the indexing value components of the thread-ID (e.g., `threadIdx.x`, `threadIdx.y`) are a virtual constant to the threads sharing the same index value (Figure 4 (c)), and are also programmatic. Compared to D-branches, branches depending on programmatic values (*P-branches*) are likely to be aligned because the (programmatic) values being used for resolving predicates are the same among threads (e.g., threads within the same warp or ones sharing a common home SIMD lane). Although P-branches that cause only partial alignment (Figure 5 (c)) can be compacted as-is, we observe that such cases are relatively rare compared to P-branches causing full alignment and preventing compaction (Figure 5 (a,b)).

It is worth mentioning that a branch condition depending on both programmatic and data value (Figure 4 (b)) behaves as a D-branch; this is because the data each thread is referencing will likely cause the predicate to be evaluated differently, regardless of the programmatic value. In Section 3.3 and Section 6, we categorize divergent branches into P-/D-branches and quantitatively verify our observations.

3.3 Key Observations and Analysis

Figure 6 summarizes the overall behavioral of branch divergence and compactability of our benchmarks. First, Figure 6 (a) shows a breakdown of all dynamically executed branches based on their divergence and potential compactability. The divergent branches are further categorized as P-/D-branches based on a taint analysis of the predicates of branch instructions using GPUOcelot [12] in Figure 6 (b) (taint analysis detailed in Section 5). In addition, we evaluate TBC_{ideal} and TBC's *compaction rate* across the divergent P-/D-branches in order to compare what fraction of the potential compaction opportunities are actually utilized (Figure 6 (c)). We define *compaction rate* as the fraction of compactable paths among all the (CTA-wide) paths generated from divergent branches (true/false) in an application.

To quantify our results, we evaluate the number of warps for each path when applying compaction with TBC or ideally. We use NW_{NoTBC} to refer to the number of warps in a CTA without compaction applied, NW_{TBC} for the number of warps after compaction, and NW_{ideal} for the minimum number of warps possible with any compaction mechanism. The lower bound on compaction is simply the minimum number of warps needed to execute the number of threads

Code #1) Branch depending on data arrays - (i)

```
0    // Code snippet from the kernel of BFS benchmark
1    // g_graph_visited and g_graph_edges are data array parameters.
2
3    int tid = blockIdx.x*MAX_THREADS_PER_BLOCK + threadIdx.x;
4
5    …
6    int id = g_graph_edges[…];
7    if( !g_graph_visited[id] )
8    {
9         …
10   }
```

(a) Threads that load a data array value (*g_graph_visited*) of zero execute the true path. The branch at line 7 is therefore data-dependent.

Code #2) Branch depending on data arrays - (ii)

```
0    // Code snippet from the kernel of SORTNW benchmark
1    // s_key[ ] and s_val[ ] are data array parameters.
2
3    uint ddd = dir & ((threadIdx.x&(size/2)) != 0);
4    …
5    Comparator(s_key[~], s_val[~], s_key[~], s_val[~], ddd);
6    …
7    __device__ inline void Comparator(uint& keyA, keyB, uint dir ){
8      if( (keyA>keyB) == dir ){ … };
9    }
```

(b) The intermediate variable (*ddd*), which is tainted by a programmatic value, is combined with values from data arrays (*s_key, s_val*) to calculate the predicate. As the data values referenced by each thread are likely different, the branch at line 8 is data-dependent.

Code #3) Branch dependent on a *programmatic* value - (i)

```
0    // Code snippet from the kernel of BITONIC benchmark
1
2    const unsigned int tid = threadIdx.x;
3    …
4    for (unsigned int k = 2; k <= NUM; k *= 2){
5      for (unsigned int j = k/2; j>0; j/=2){
6        unsigned int ixj = tid ^ j;
7        if( ixj > tid ) {
8          if( (tid & k)==0 ){…} else {…}
9        }
10       __syncthreads();
11     }
12   }
```

(c) The index value of a thread-ID (*tid*) solely determines whether the true path (*if*) or the false path (*else*) is taken. Hence, the branches at line 7 and 8 are programmatic.

Code #4) Branch dependent on *programmatic* values - (ii)

```
0    // Code snippet from the kernel of Mandelbrot benchmark
1    // imageW and imageH are scalar input parameters of the kernel
2
3    const int ix = blockDim.x * blockX + threadIdx.x;
4    const int iy = blockDim.y * blockY + threadIdx.y;
5    …
6    if( (ix < imageW) && (iy < imageH) )
7    {
8         …
9    }
```

(d) An intermediate variable (*ix, iy*), which is tainted by programmatic values, is combined with another programmatic value (*imageW, imageH*) from a scalar input parameter of the kernel. The branch at line 6 is therefore programmatic.

Figure 4: Example kernel codes containing P-branches and D-branches.

in each path of the CTA:

$$NW_{ideal} = \lceil \frac{NumActive_{CTA}}{SIMD_{width}} \rceil \quad (1)$$

where, $NumActive_{CTA}$ refers to the total number of threads active at the diverging path across the CTA and $SIMD_{width}$ designates the width of the SIMD pipeline. When NW_{ideal} and NW_{TBC} (number of warps after compaction with TBC) for a path are both smaller than the number of warps without compaction (NW_{NoTBC}), then the path can be compacted as-is with TBC. When NW_{NoTBC} and NW_{TBC} are equal and NW_{ideal} is smaller, a path is potentially compactable but not be compacted as-is with TBC (Figure 5 (a)). When the values of NW_{ideal} and NW_{NoTBC} are the same, a path has neither potential nor actual compactability (Figure 5 (b)).

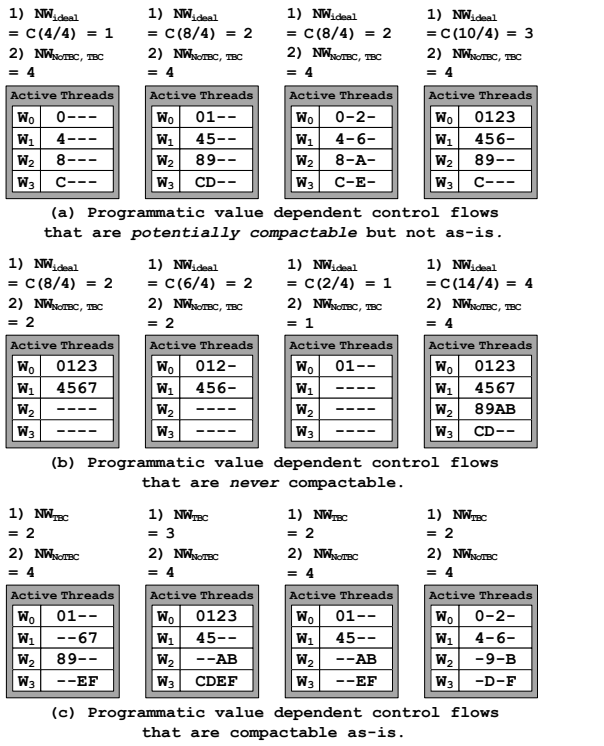


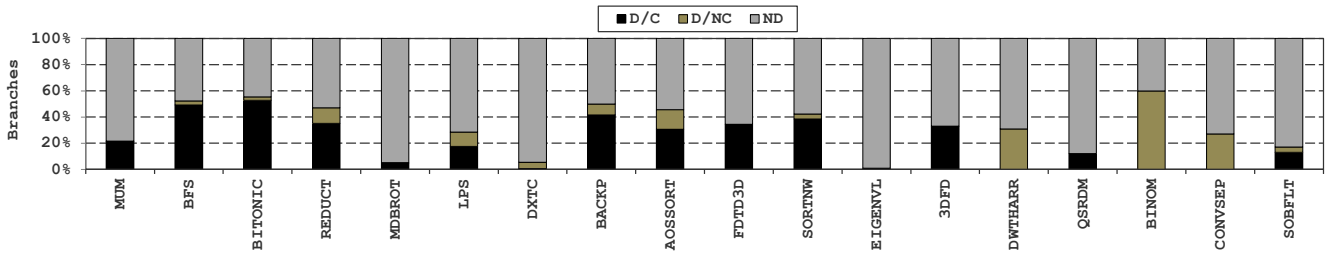
Figure 5: Active warp status for programmatic-value dependent control flow. W_n designates the warp with warp-ID n.

By definition, benchmarks with a nonzero TBC compaction rates in either P-/D-branches (Figure 6 (c)) exhibit improvements in $SIMD_{util}$ with TBC. As expected, all 9 benchmarks containing any D-branches exhibit nonzero TBC compaction rates for this branch type, and we confirm our intuition that previous compaction mechanisms work relatively well for D-branches. However, in 13 of the 15 benchmarks containing P-branches, no compaction occurs at all, as seen by their zero P-branch TBC compaction rate in Figure 6 (c). Of these 13 benchmarks, only 5 have no potential for compaction (zero compaction with TBC_{ideal}). The other 8 benchmarks (BITONIC, REDUCT, LPS, BACKP, AOSSORT, FDTD3D, 3DFD, and QSRDM) show that substantial opportunity for improving $SIMD_{util}$ is untapped with TBC and other current compaction techniques because a thread's home SIMD lane is fixed. We tackle such P-branches and enable new compaction opportunities using SIMD lane permutation, which we describe in the following section.

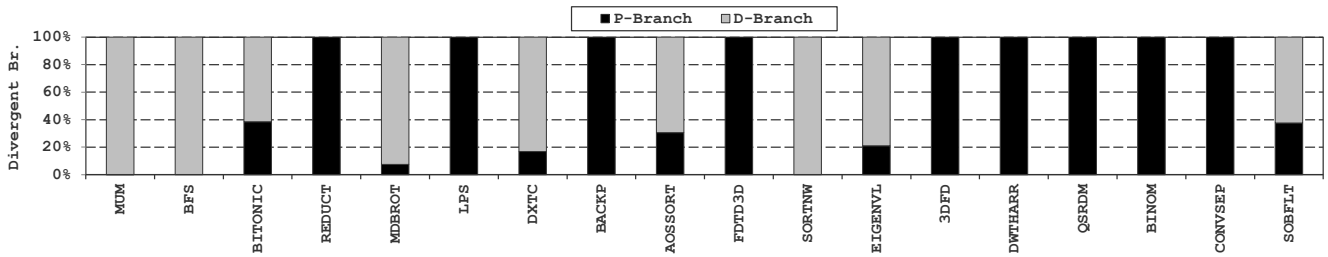
4. SIMD LANE PERMUTATION

4.1 Motivation

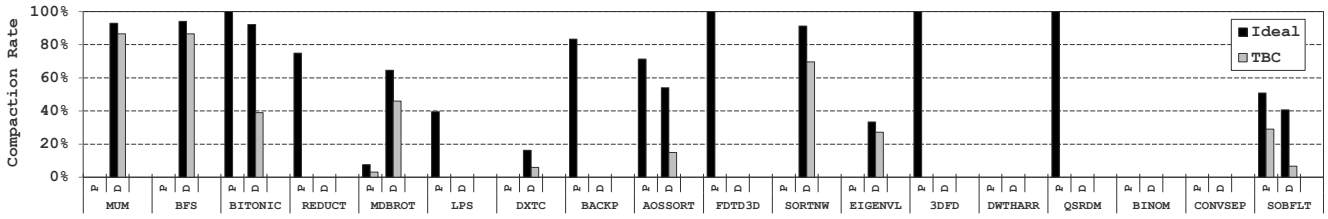
SIMD lane permutation (SLP) is based on the insight that if the home SIMD lanes of threads are permuted from their



(a) Breakdown of all dynamically executed conditional branches into: (i) non-divergent branches (ND), (ii) divergent branches with no potential compactability (D/NC), and (iii) divergent branches with potential compactability (D/C); potentially-compactable branches are those that can be compacted with TBC_{ideal}.



(b) Breakdown of divergent branches (D/NC and D/C in (a)) into P-branches and D-branches.



(c) Compaction rate of P-branches and D-branches using TBC_{ideal} and TBC. Note that because all divergent branches of DWTHARR, BINOM, and CONVSEP are never compactable (D/NC), the corresponding TBC_{ideal} compaction rates are zero.

Figure 6: Analysis of the behavior of control divergence and compactability.

sequentially-assigned locations, the alignment of threads can be eliminated in many cases. The permutation is applied to each thread when a warp is launched, before it starts executing, and is then fixed until the warp completes execution. This improves compactability while maintaining the cost-efficiency of the baseline compaction mechanism and SIMT architecture.

Permuting the home SIMD lane of each thread requires changing how each thread-ID is mapped to a SIMD lane. Figure 7 illustrates some example permutations and their associated mapping, including the ones discussed in prior work [9, 15]. Because these mapping functions can be calculated statically using only the thread-IDs and warp size, the compaction-hardware requires no modification. While older NVIDIA GPUs mandated threads to access memory in sequence to enable memory coalescing and minimize memory transactions, AMD GPUs and recent NVIDIA GPUs (starting with NVIDIA Compute Capability 2.0) can coalesce any collection of addresses into the minimum possible number of transactions [28]. Thus, thread order within a warp does not impact performance and SLP can be incorporated smoothly without disrupting memory access behavior.

Some expert programmers of current GPUs occasionally rely on undocumented-behavior for optimization and tuning. SLP may break applications that rely on the fact that in current GPU implementations the same logical thread location within a warp always wins arbitration when resource conflicts between threads in a warp occur. To preserve this arbitration behavior, which may be very desirable in some cases, a programmer can explicitly disable SLP on a particular kernel. Note that compaction itself already breaks this particular undocumented behavior.

4.2 Pitfalls of a Random Permutation

While the permutations in Figure 7 (b)-(h) can all break the alignment of active lanes in some cases, their effectiveness in increasing compactability can vary substantially because some permutations are only optimal for certain divergence patterns. *Odd_Even* [15], for example, works most effectively for the alignment pattern shown in Figure 8 (c), but provides no benefit when active threads are grouped together as in Figure 8 (b) (active/active, inactive/inactive). Another permutation, such as *XOR*-ing only the odd-ID warps with $\left(\frac{SIMD_{width}}{2}\right)$, on the other hand, will perfectly compact all the threads in Figure 8 (b), but none for Figure 8 (c).

Incorporating randomness in the permutation function (e.g., *Random*, *WID*, and *Rev_WID* in Figure 7 (f–h)) intuitively seem more effective in redistributing clustered threads. However, such random permutations do not perform well when a large fraction of the threads are active across the CTA. For example, consider a case where half the threads in each

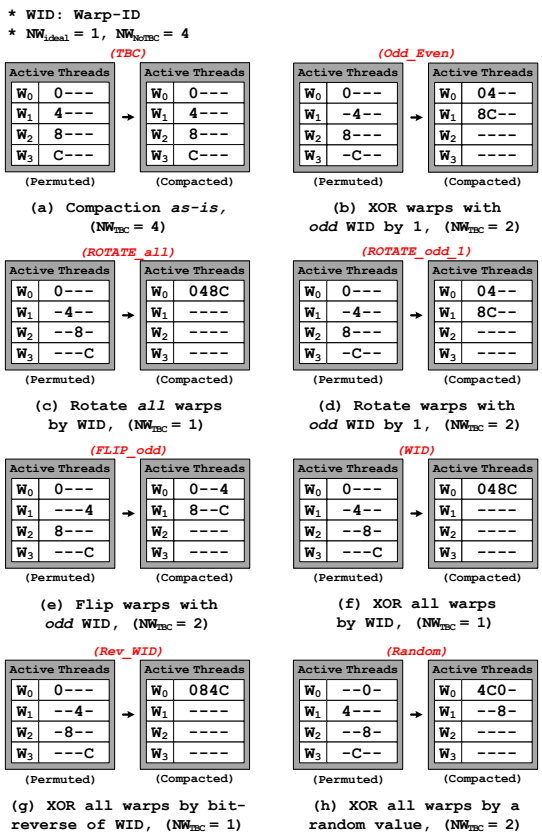


Figure 7: Examples of various SIMD lane permutation mechanisms that alter the alignment of active threads to lanes.

warp are active – unless the active threads are permuted exactly to the other half of the (vacant) lanes, compaction will fail. Figure 8 shows an example of a P-branch causing half of the threads in a CTA to be active. This behavior is exhibited in applications such as BITONIC, QSRDM, MD-BROT, and others. While such a divergence pattern can occur from an under-optimized kernel, it can also be generated by the nature of the algorithm itself. Note that *Odd_Even* and *Rev_WID* both only effectively compact one of the two example patterns but not both (Figure 9). In general, we observe that previously discussed permutation functions fall short of ideal and are not robust because they are based on empirical observations of a subset of divergence patterns.

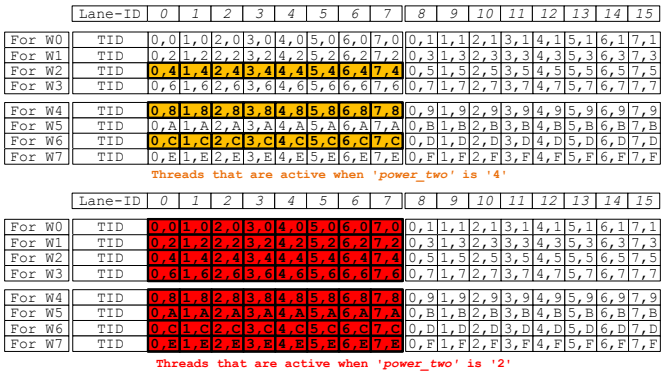
4.3 Balanced Permutation

Based on the previous discussion, we construct a robust permutation mechanism. An important insight is that the permutation to be applied should distribute active threads across the SIMD lanes in a *balanced* manner because aligned divergence from P-branches frequently exhibits highly skewed distributions of active lanes. *Balanced* is designed such that for any CTA (with fewer than $SIMD_{width}$ warps) each physical lane only has a single instance of each logical thread location within a warp. This characteristic is unique to *Balanced* and provides its robustness. Intuitively, threads within warps with an even warp-ID (WID) are only permuted within each half-warp (each warp uses a different *XOR* mask). Every even-ID warp is paired with an odd-ID warp that uses a complementary mask and ensures that

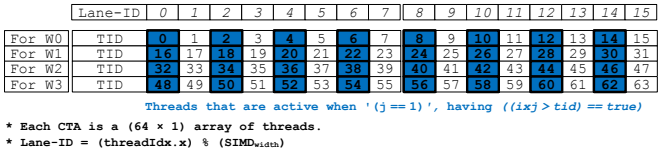
Code #5) Programmatic branch causing only the 1st half of the warp active

```
0    // Code snippet from the kernel of BACKP benchmark
1    // CTA is a (8 × 16) 2-D array of threads.
2
3    int tx = threadIdx.x;
4    int ty = threadIdx.y;
5    ...
6    for (int i=1; i<=__log2f(HEIGHT); i++){
7      int power_two = __powf(2,i);
8
9      if( ty % power_two == 0 ) {...}
10     ...
11   }
```

(a) Code snippet from BACKP benchmark that exhibits programmatic divergence.



(b) Active threads with value-dependent lane alignment, corresponding to the branch at line 9 of (a) (dependent on *power_two*).



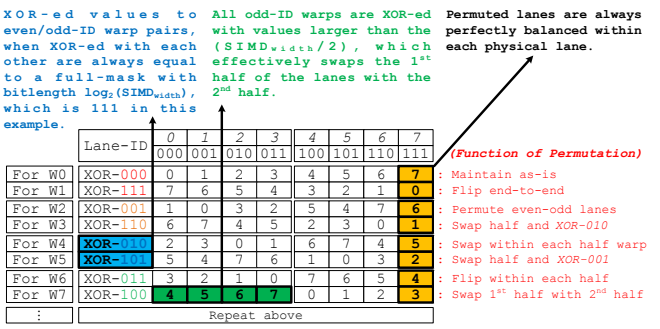
(c) Active threads with programmatic lane alignment, corresponding to line 7 of Figure 4 (c)

Figure 8: Examples of aligned divergence patterns in Figure 9 (b–c).

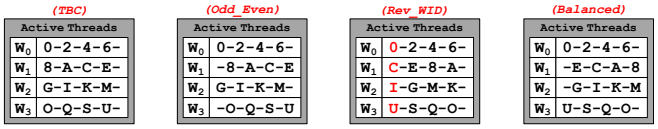
threads are shuffled to the second half of each warp. The permutation algorithm is to *XOR* the logical thread location with a mask that is computed differently for each warp using the warp ID (WID). The *Balanced* permutation masks are computed for even warp IDs using the formula in Equation 2. The masks for odd warp IDs are a bit-wise inverse of their even warp ID pairs.

$$XOR_{evenWID} = \frac{evenWID}{2} \tag{2}$$

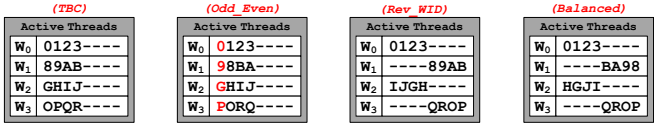
Figure 9 illustrates the functionality of SLP with the proposed *Balanced* permutation algorithm. Without SLP, lane-ID 0 is always assigned to physical lane 0, for example. With *Balanced*, on the other hand, each physical lane only has a single instance of each lane-ID. This *vertical balance* is unique to the carefully-designed *Balanced* permutation and is clearly shown in the highlighted physical lane 7 in Figure 9. Figure 9 also illustrates the overall construction of *Balanced*. Randomized permutations only achieve this balance on average, while any individual CTA is likely to be somewhat imbalanced. This imbalance is greater on aver-



(a) Proposed *(Balanced)* Algorithm (assuming WarpSize and $SIMD_{width}$ of 8).



(b) Example control flow where *(Rev_WID)* does not compact well.



(c) Example control flow where *(Odd_Even)* does not compact well.

Figure 9: Proposed *Balanced* permutation.

age for CTAs that have a small number of warps. *Balanced* works well even for these CTAs, because of the even/odd complementary masks it uses.

We present a detailed quantitative evaluation of various permutations in Section 6, but in summary, *Balanced* is very robust and is either the most effective permutation, or is within 99.4% of the best permutation in all our experiments. Because *Balanced* tends to outperform other permutations, is within a small fraction of the best permutation in the rare cases it is not already the best, and rarely degrades performance, we propose to always apply *Balanced* when allocating a new warp.

4.4 Implementation of SLP

Enabling SLP on top of TBC requires storage for the permutation table and a small amount of logic for permuting the home SIMD lanes. The storage requirements for *Balanced* are just 5 bits for each of the 32 unique *XOR* masks, totaling 160 bits per SM.

The required logic is just a handful of *XOR* gates per lane. Prior work [14, 31] discusses the hardware implementation of the compaction mechanisms in detail, which concluded that the overhead is less than $1mm^2$ for an entire chip in $65nm$ technology. In addition, a single control bit is needed to allow a programmer to disable SLP for a kernel to maintain backward compatibility for codes that rely on the undocumented behavior of a single arbitration order between logical threads within a warp.

4.5 Software-Based Permutation

Zhang et al. [40] propose *reference redirection*, which is a software only technique that attempts to provide an alternative to hardware compaction. Reference redirection

Pseudo PTX Program) Taint analysis

```
0    ...
1    mov       %r2, %tid.x     // %r2 is tainted by a programmatic value
2    setp.eq   %p1, %r2, 0     // Predicate register %p1 tainted from %r2
3    @%p1 bra  [JUMP_1]        // Branch depends on programmatic value
4    ld.param  %r3, [g_data]   // Pointer to a data array is loaded to %r3
5    add       %r4, %r2, %r3   // %r4 contains the address to load from
5    ld.global %r5, [%r4]      // %r5 is tainted by data array value
6    setp.gt   %p6, %r5, 1     // Predicate register %p6 tainted from %r5
7    @%p6 bra  [JUMP_2]        // Branch depends on data value
8    mov       %r2, 100        // Taint is cleared by an immediate value
9    ...
```

```
* tid.x   : x-index value of a thread-ID
* g_data  : data array allocated at global memory
```

Figure 10: Pseudo PTX code explaining the methodology of the taint analysis.

Table 2: Microarchitectural configuration of GPGPU-Sim.

Number of SMs	30
Threads per SM	1024
Threads per warp	32
Width of SIMD pipeline	32
Registers per SM	16384
Shared memory per SM	32KB
Warp scheduling policy	Sticky Round-Robin
L1 Cache (size/assoc/line)	32KB/8-way/64B
L2 Cache (size/assoc/line)	1024KB/64-way/64B
Memory controller	Out-of-order (FR-FCFS)
Number of memory channels	8

statically rearranges threads with similar control flow into a common warp to mitigate the impact of control divergence. This software technique often incurs significant overhead for two main reasons. First, rearranging the threads has a runtime component that must be amortized over long-running threads and kernels. Second, the new thread groups can degrade memory coalescing behavior for the entire duration of the kernel. The key difference from hardware compaction is that reference redirection is static for the entire kernel, whereas compaction only dynamically rearranges threads in those basic blocks where it is effective. Thus, while reference redirection can degrade the memory performance of the entire kernel, hardware compaction executes identically to baseline in basic blocks that do not diverge. A direct comparison between reference redirection [40] and SLP is beyond the scope of this paper and we leave it to future work.

5. EVALUATION METHODOLOGY

We evaluate the effectiveness of SLP using a combination of GPUOcelot (r1865) [12] and GPGPU-Sim (version 2.1) [1, 7]. GPUOcelot is an open source compiler infrastructure supporting NVIDIA's Parallel Thread Execution ISA (PTX) version 3.0; we use its PTX emulator to classify divergent branches into P-/D-branches using taint analysis (Figure 10). We also leverage the CTA-wide active mask information in the emulator to analyze compaction rate (Section 6.1) and $SIMD_{util}$. We ran all benchmarks from those included in the CUDA-SDK [27], the Rodinia [10] suite, and those provided with GPGPU-Sim [7, 16, 17, 18, 19, 34], which could be executed by GPUOcelot. Of these 40 benchmarks, we only show and discuss the results of the 18 benchmarks that exhibited branch divergence, excluding 22 benchmarks that have $SIMD_{util}$ of over 99%. The benchmarks are summarized in Table 1.

In addition to GPUOcelot, we use GPGPU-Sim to evalu-

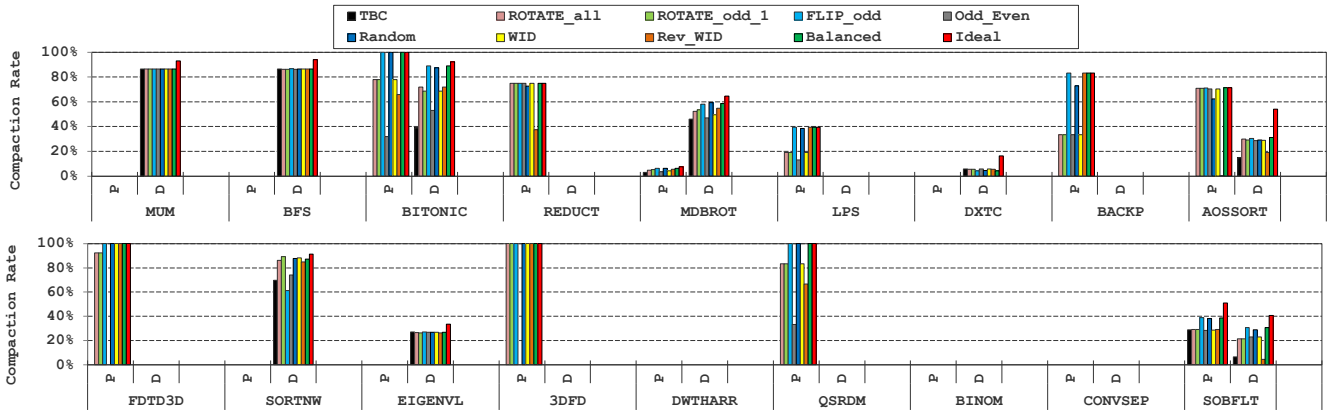


Figure 11: Compaction rate per branch type with different permutations.

ate the impact of SLP on overall performance (Section 6.3). GPGPU-Sim is a publicly available, cycle-level performance simulator of a general purpose GPU that supports PTX version 1.4. To compare SLP's effectiveness with the state-of-the-art compaction mechanisms, we implement CAPRI [31] in GPGPU-Sim. CAPRI was shown to consistently outperform TBC in terms of performance, so we only report the results of CAPRI in Section 6.3. Other key microarchitectural aspects of the simulator are summarized in Table 2 [2]. GPGPU-Sim 2.1 does not support some runtime APIs, such as OpenGL, which are supported by GPUOcelot. We therefore only report the performance results of the 8 out of 18 benchmarks that GPGPU-Sim 2.1 can simulate without modification. All CUDA applications were compiled as-is and with the parameters provided with GPGPU-Sim and GPUOcelot.

6. EXPERIMENTAL RESULTS

In this section we detail the evaluation results of SLP, including the enhancements in compactability, SIMD lane utilization, and performance. We assume TBC as the baseline compaction mechanism for compactability (Section 6.1) and $SIMD_{util}$ (Section 6.2) so that all compactable branches are considered. For performance (Section 6.3), on the other hand, we use CAPRI as the baseline compaction architecture as it consistently outperforms TBC in terms of execution time, but skips some compaction opportunities by design.

6.1 Compactability

Figure 11 shows the compaction rate of P-branches and D-branches achieved with different permutations (including baseline TBC) across our 18 benchmarks. Looking at P-branches first, 10 of the 15 benchmarks contain compactable P-branches (see Section 3.3). Despite their heuristic nature, all but one of the permutations significantly improve P-branch compactability over the 3.2% compaction rate possible without SLP; SLP with *Odd_Even* cannot compact FDTD3D and 3DFD. However, only SLP with our carefully designed *Balanced* permutation comes close to ideal compaction for P-branches, averaging a compaction rate of 71.5% – 98% of the ideal 72.7% average compaction rate. *Balanced* precisely matches *Ideal* in 7 of the 10 benchmarks. In FDTD and MDBROT, *Balanced* is within 1.1% of ideal. SOBFLT is the only benchmark we evaluated in which *Balanced* failed to to achieve near-ideal compaction rate (38.7% compared to 50.9%). *Balanced* was still within 1% of the best permutation (*FLIP_odd*) even in this benchmark. In contrast, the previously proposed permutations, *Odd_Even* and *Rev_WID*, only achieve an average of 28.9% and 52.8% respectively. *Odd_Even* only works for a very specific pattern and is not robust across the applications. Randomized permutations do not perform well when there is a large fraction of active threads, which we elaborate on below.

Figure 12 shows a breakdown of the compactability of P-branches depending on the fraction of threads that are active in the CTA after the branch point. The randomized *Rev_WID* [9] permutation is more effective than the naive *Odd_Even*, but *Rev_WID* still misses significant opportunities for compaction when half or more of threads are active (e.g., in BITONIC, REDUCT, MDBROT, AOSSORT and QSRDM). In fact, *Rev_WID* even performs worse than *Odd_Even* for REDUCT and AOSSORT. *Balanced* does well even in these challenging cases, missing few or zero opportunities for compaction (small to insignificant yellow, orange, or red portions in the figure).

Of the 18 benchmarks with divergent branches, 9 have divergent D-branches. As expected, baseline TBC does much better compacting D-branches than P-branches and has an average compaction rate of 42.5% compared to the ideal average of 64.4%. While the average compactability with the baseline TBC is reasonable overall, it is quite poor in 5 of the 9 benchmarks (BITONIC, MDBROT, AOSSORT, SORTNW, and SOBFLT), with an average of just 35.3%. SLP significantly improves compaction in these 5 benchmarks. *Balanced* provides the most improvement and shows its robustness by averaging 59.3% – 86% of the ideal 68.5% average compaction in these 5 benchmarks. In fact, *Balanced* always achieve more than 97.9% of the best permutation in all benchmarks except DXTC, for which applying no permutation is best. The active threads in DXTC are exhibited in groups of clusters, but the groups themselves are randomly scattered across the CTA which makes SLP less effective. As with P-branches, *Odd_Even* was the worst performer and even trailed the average of baseline TBC.

Figure 13 shows the compactability breakdown depending on fraction of active threads for D-branches. While the benefits are not as pronounced as with P-branches, *Balanced* again demonstrates that random permutations are a poor choice when a large fraction of threads are active. Note that with D-branches, SLP does slightly impair compactability in a few cases, but all are within 1% of baseline TBC. One possible optimization to prevent degrading baseline is to utilize

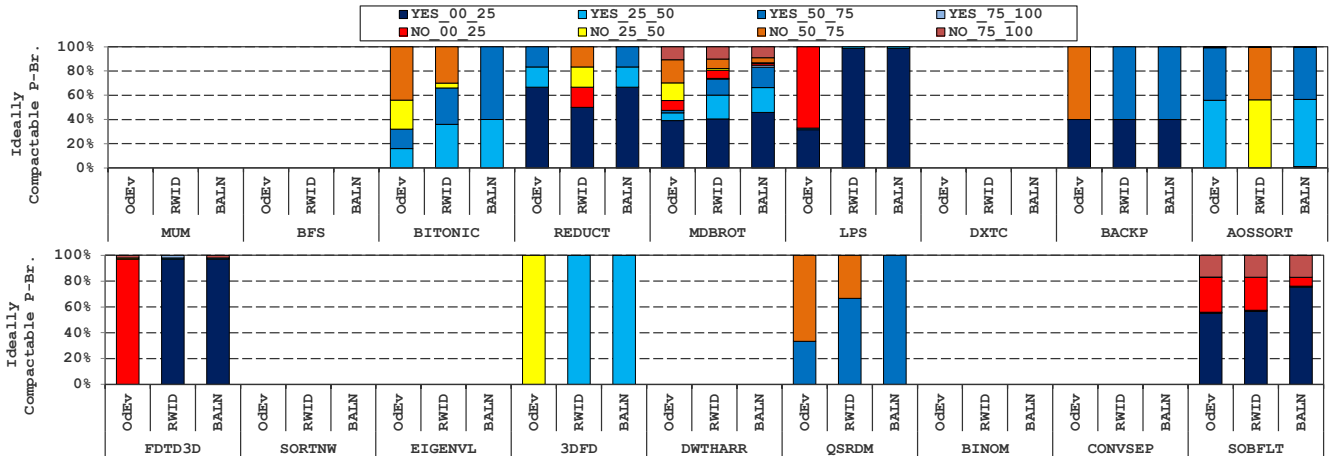


Figure 12: Breakdown of P-branch compaction rate by fraction of active threads across the CTA at the branch point. *(XX_YY_ZZ)* designates whether a branch is compacted or not (*XX*) for a particular fraction range of active threads in that path (*YY* <= ActiveThreads < *ZZ*). Note that the full 100% bar is equivalent to the *Ideal* bar in Figure 11. *OdEv*, *RWID*, *BALN* refers to *Odd_Even*, *Rev_WID*, and *Balanced* respectively.

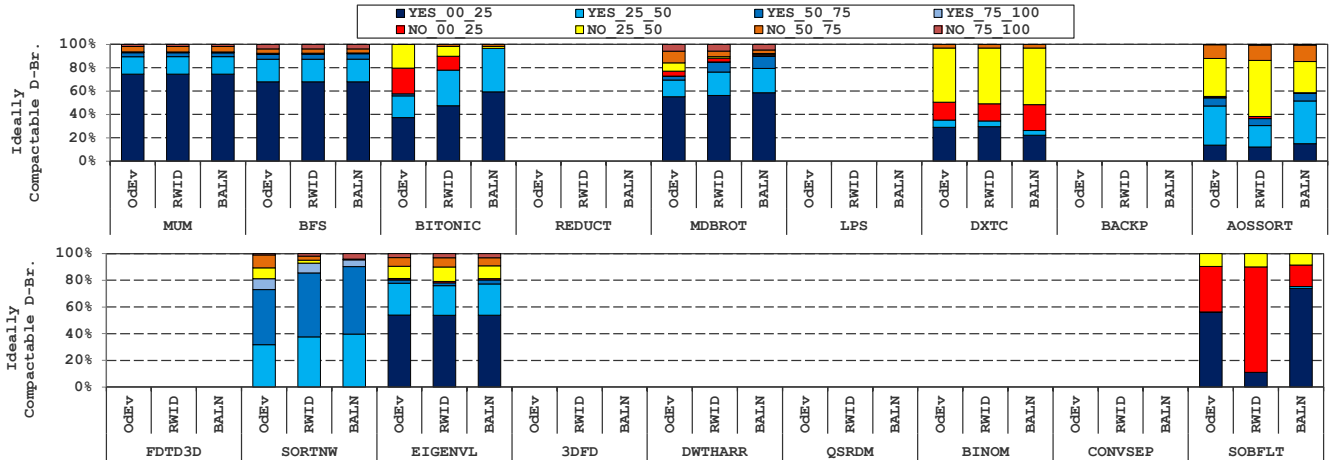


Figure 13: Breakdown of D-branch compaction rate by fraction of active threads in a path across the CTA.

compiler-support; SLP is disabled when the percentage of D-branches is above a threshold. We leave the exploration of such an option to future work.

6.2 SIMD Lane Utilization

Our definition of compaction rate ignores the magnitude of compaction and focuses solely on whether any reduction in the number of warps was achieved. In contrast, $SIMD_{util}$ ignores how often compaction was successful and instead represents the overall magnitude of compaction. As shown in Figure 14, for instance, although there are multiple permutation methods that achieve the full compaction rate for BITONIC, REDUCT, FDTD3D, and 3DFD, the resulting $SIMD_{util}$ using these permutations is highly variable and only *Balanced* offers consistently high improvements. *Balanced* most effectively reduces the number warps overall, with average increases of 11.3% over the *No_TBC* baseline and 7.1% (max 34%) over TBC without SLP. *Balanced* is also either the permutation with the highest $SIMD_{util}$ (in 7 benchmarks) or within 99.4% of the best performing permutation in the other benchmarks. *Odd_Even* provides the smallest benefits, but still increases $SIMD_{util}$ by 6% and 2.1% (max 18%) compared to *No_TBC* and TBC respectively. *Rev_WID*, which is the second previously discussed permutation [9], does better than *Odd_Even* with an average 8.8% and 4.8% increase over *No_TBC* and TBC respectively. However, *Rev_WID* lacks robustness and significantly trails the best performing permutation by up to 14.3% (e.g., for BITONIC, REDUCT, AOSSORT, QSRDM). Among the 5 benchmarks that experienced a lowered compaction rate for their D-branches with SLP compared to baseline, all also experience a decrease in $SIMD_{util}$ for a some SLP permutations but of less than 0.1%.

6.3 Overall Performance

Figure 15 shows the performance of those 8 benchmarks that can be executed as-is in GPGPU-Sim 2.1. The SLP mechanisms are implemented on top of CAPRI and the associated performance results are shown relative to those of *No_TBC* and CAPRI. *Balanced* provides the highest average IPC increase, outperforming baseline by 11.6% and CAPRI by 7% (both harmonic means). *Balanced* also exhibits the maximum speedup observed, improving BITONIC by 25.6% and BACKP by 15.2%. It is also the best performing SLP permutation on 4 of the 8 benchmarks and is always within 98.9% of the best permutation. In contrast, the second-best

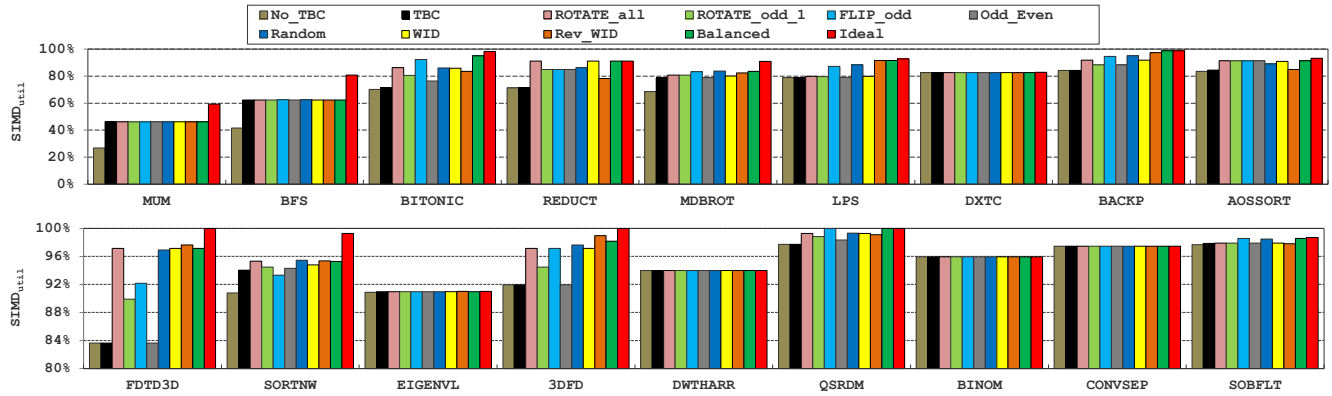


Figure 14: Average SIMD lane utilization with different permutations. Note that scale of the bottom chart above is $80 - 100\%$.

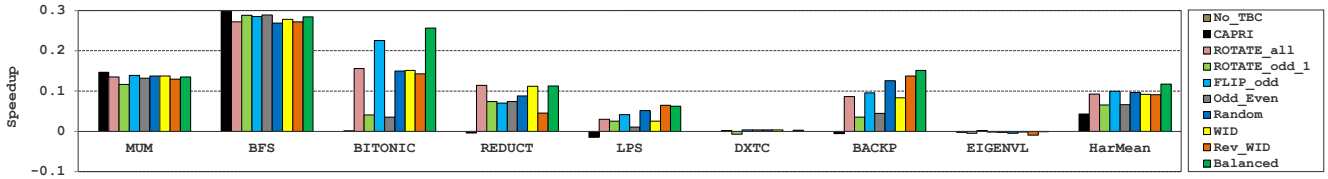


Figure 15: Speedup of compaction with different permutations over no-compaction.

permutation, *FLIP_odd*, achieves an average speedup of 5% over CAPRI and in worst case achieves only 90.5% of the best performing SLP.

Because SLP degrades the compaction rate and $SIMD_{util}$ of MUM, BFS, DXTC, and EIGENVL, performance for these benchmarks decreased. *Balanced* and *FLIP_odd* showed the least degradation in performance (0.3% and 0.4% degradation compared to CAPRI, respectively). *Rotate_odd_1* caused the greatest average IPC degradation for these benchmarks (1.1%), but overall, even this permutation improved performance by 2.1% (harmonic mean). Although we do not evaluate the performance of the other 10 benchmarks, we expect the strong correlation between $SIMD_{util}$ and performance to apply to them as well.

6.4 Impact on Memory System

Compaction involves dynamically rearranging threads from different warps and scheduling them together at the same time. While SLP itself does not disturb the memory coalescing capability (as discussed in Section 4.1), the dynamic formation of warps through compaction can degrade memory access behavior compared to a pipeline that does not compact at all. Among the evaluated benchmarks, those exhibiting a substantial increase in $SIMD_{util}$ and performance did show a noticeable increase in L1 misses, with an average increase of 6.9% and a maximum increase of 20% (in BFS). Although L1 misses were more frequent than without compaction, the impact on L2 misses and memory traffic is negligible (1.3% average increase). These results are in line with the observations from prior work [14, 25, 31]. Because the benefits of compaction outweigh the increase in L1 traffic, overall performance is increased, as previously discussed.

7. CONCLUSIONS

In this paper we argue that previous SIMT compaction mechanisms, which seek to alleviate the detrimental impact of control divergence, fall short because of their limited applicability. We explain and quantitatively demonstrate that in many cases, the way threads are associated with SIMD lanes causes aligned divergence patterns that prevent compaction. Our detailed analysis reveals that such alignment mainly originates from non-data dependent, programmatic, branches. Although diverging paths from these programmatic branches do not compact well as-is (an average of 3.2% compaction rate), there is substantial opportunity (72.7% compaction rate) if the fixed association of threads to lanes can be relaxed. Importantly, such programmatic branches are common in applications with irregular control (11 of the 18 benchmarks we evaluate).

We propose and evaluate SLP, which expands the applicability of compaction by reducing, and even eliminating, aligned divergence. SLP permutes the mapping of logical thread locations to physical SIMD lanes when a warp is launched. This breaks aligned divergence patterns resulting from conditionals that depend only on programmatic values. We construct a novel and robust *Balanced* permutation technique, which enables an average of 71.5% of programmatic branches to be compacted – 98% of the ideal 72.7%. As a result, SLP with *Balanced* achieves the highest $SIMD_{util}$ and performance of all compaction and permutation mechanisms across the 18 benchmarks we study.

Because the permutation scheme has minimal hardware overhead and does not directly impact any component other than determining the association of threads and lanes, we argue that it should become the default architecture for GPUs that utilize thread compaction.

8. ACKNOWLEDGEMENTS

We thank the developers of GPGPU-Sim and GPUOcelot. We also thank the anonymous reviewers, who provided excellent feedback for preparing the final version of this paper.

9. REFERENCES

[1] GPGPU-Sim. http://www.gpgpu-sim.org.
[2] GPGPU-Sim Manual. http://www.gpgpu-sim.org/manual.

[3] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '83, pages 177–189, New York, NY, USA, 1983. ACM.

[4] AMD Corporation. AMD Radeon HD 6900M Series Specifications, 2010.

[5] AMD Corporation. ATI Stream Computing OpenCL Programming Guide, August 2010.

[6] T. Anderson, S. Owicki, J. Saxe, and C. Thacker. High-speed Switch Scheduling for Local-Area Networks. In *ACM Transactions on Computer Systems (TOCS)*, 1993.

[7] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS-2009)*, April 2009.

[8] W. Bouknight, S. Denenberg, D. McIntyre, J. Randall, A. Sameh, and D. Slotnick. The Illiac IV System. In *Proceedings of the IEEE*, volume 60, pages 369–388, April 1972.

[9] N. Brunie, S. Collange, and G. Diamos. Simultaneous Branch and Warp Interweaving for Sustained GPU Performance. In *39th International Symposium on Computer Architecture (ISCA-39)*, June 2012.

[10] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IEEE International Symposium on Workload Characterization (IISWC-2009)*, October 2009.

[11] G. Diamos, B. Ashbaugh, S. Maiyuran, A. Kerr, H. Wu, and S. Yalamanchili. SIMD Re-Convergence At Thread Frontiers. In *44th International Symposium on Microarchitecture (MICRO-44)*, December 2011.

[12] G. Diamos, A. Kerr, S. Yalamanchili, and N. Clark. Ocelot: A Dynamic Compiler for Bulk-Synchronous Applications in Heterogenous Systems. In *19th International Conference on Parallel Architecture and Compilation Techniques (PACT-19)*, September 2010.

[13] D. Ferguson, Y. Yemini, and C. Nikolaou. Microeconomic Algorithms for Load Balancing in Distributed Computer Systems. In *8th International Conference on Distributed Computing Systems*, 1988.

[14] W. W. Fung and T. M. Aamodt. Thread Block Compaction for Efficient SIMT Control Flow. In *17th International Symposium on High Performance Computer Architecture (HPCA-17)*, February 2011.

[15] W. W. Fung, I. Sham, G. Yuan, and T. M. Aamodt. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *40th International Symposium on Microarchitecture (MICRO-40)*, December 2007.

[16] A. Gharaibeh and M. Ripeanu. Size Matters: Space/Time Tradeoffs to Improve GPGPU Applications Performance. In *2010 International Conference for High Performance Computing, Networking, Storage and Analysis (SC-2010)*, November 2010.

[17] M. Giles. Jacobi iteration for a Laplace discretisation on a 3D structured grid. http://people.maths.ox.ac.uk/gilesm/hpc/NVIDIA/laplace3d.pdf, 2008.

[18] M. Giles and S. Xiaoke. Notes on using the NVIDIA 8800 GTX graphics card. http://people.maths.ox.ac.uk/ gilesm/hpc/, 2008.

[19] P. Harish and P. Narayanan. Accelerating Large Graph Algorithms on the GPU Using CUDA. In *High Performance Computing HiPC 2007*, volume 4873, pages 197–208. 2007.

[20] U. Kapasi, W. Dally, S. Rixner, P. Mattson, J. Owens, and B. Khailany. Efficient conditional operations for data-parallel architectures. In *33th International Symposium on Microarchitecture (MICRO-33)*, December 2000.

[21] B. Li and A. Eryilmaz. Exploring the Throughput Boundaries of Randomized Schedulers in Wireless Networks. In *IEEE/ACM Transactions on Networking (TON)*, 2012.

[22] W. Lin, S. Reinhardt, and D. Burger. Reducing DRAM Latencies with an Integrated Memory Hierarchy Design. In *7th International Symposium on High Performance Computer Architecture (HPCA-7)*, February 2001.

[23] L. Liu, Z. Cui, M. Xing, Y. Bao, M. Chen, and C. Wu. A Software Memory Partition Approach for Eliminating Bank-Level Interference in Multicore Systems. In *21st International Conference on Parallel Architecture and Compilation Techniques (PACT-21)*, September 2012.

[24] J. Meng, D. Tarjan, and K. Skadron. Dynamic warp subdivision for integrated branch and memory divergence tolerance. In *37th International Symposium on Computer Architecture (ISCA-37)*, 2010.

[25] V. Narasiman, C. Lee, M. Shebanow, R. Miftakhutdinov, O. Mutlu, and Y. Patt. Improving GPU Performance via Large Warps and Two-Level Warp Scheduling. In *44th International Symposium on Microarchitecture (MICRO-44)*, December 2011.

[26] NVIDIA Corporation. NVIDIA's Next Generation CUDA Compute Architecture: Fermi, 2009.

[27] NVIDIA Corporation. CUDA C/C++ SDK CODE Samples, 2011.

[28] NVIDIA Corporation. NVIDIA CUDA Programming Guide, 2011.

[29] O. Pearce, T. Gamblin, B. Supinski, M. Schulz, and N. Amato. Quantifying the Effectiveness of Load Balance Algorithms. In *26th ACM International Conference on Supercomputing*, 2012.

[30] B. Rau. Pseudo-Randomly Interleaved Memory. In *18th International Symposium on Computer Architecture (ISCA-18)*, June 1991.

[31] M. Rhu and M. Erez. CAPRI: Prediction of Compaction-Adequacy for Handling Control-Divergence in GPGPU Architectures. In *39th International Symposium on Computer Architecture (ISCA-39)*, June 2012.

[32] M. Rhu and M. Erez. The Dual-Path Execution Model for Efficient GPU Control Flow. In *19th International Symposium on High-Performance Computer Architecture (HPCA-19)*, February 2013.

[33] R. M. Russell. The CRAY-1 computer system. *Commun. ACM*, 21:63–72, January 1978.

[34] M. Schatz, C. Trapnell, A. Delcher, and A. Varshney. High-throughput sequence alignment using graphics processing units. *BMC Bioinformatics*, 8(1):474, 2007.

[35] D. Shah, P. Giaccone, and B. Prabhakar. Efficient Randomized Algorithms for Input-Queued Switch Scheduling. In *IEEE Micro*, 2002.

[36] J. E. Smith, G. Faanes, and R. Sugumar. Vector instruction set support for conditional operations. In *27th International Symposium on Computer Architecture (ISCA-27)*, 2000.

[37] Steven Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[38] D. Tarjan, J. Meng, and K. Skadron. Increasing memory miss tolerance for SIMD cores. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC-09)*, 2009.

[39] H. Wu, G. Diamos, S. Li, and S. Yalamanchili. Characterization and Transformation of Unstructured Control Flow in GPU Applications. In *1st International Workshop on Characterizing Applications for Heterogeneous Exascale Systems*, June 2011.

[40] E. Zhang, Y. Jiang, Z. Guo, and X. Shen. Streamlining GPU Applications On the Fly. In *24th International Conference on Supercomputing (ICS'24)*, June 2010.

[41] Z. Zhang, Z. Zhu, and X. Zhang. A Permutation-Based Page Interleaving Scheme to Reduce Row-Buffer Conflicts and Exploit Data Locality. In *33rd International Symposium on Microarchitecture (MICRO-33)*, December 2000.