BOW: Breathing Operand Windows to Exploit Bypassing in GPUs

Hodjat Asghari Esfeden*, Amirali Abdolrashidi*, Shafiur Rahman*, Daniel Wong†, Nael Abu-Ghazaleh*†
* Department of Computer Science and Engineering
† Department of Electrical and Computer Engineering
University of California Riverside
{hasgh001,aabdo001,mrahm008,danwong,naelag}@ucr.edu

***Abstract*—The Register File (RF) is a critical structure in Graphics Processing Units (GPUs) responsible for a large portion of the area and power. To simplify the architecture of the RF, it is organized in a multi-bank configuration with a single port for each bank. Not surprisingly, the frequent accesses to the register file during kernel execution incur a sizeable overhead in GPU power consumption, and introduce delays as accesses are serialized when port conflicts occur. In this paper, we observe that there is a high degree of temporal locality in accesses to the registers: within short instruction windows, the same registers are often accessed repeatedly. We characterize the opportunities to reduce register accesses as a function of the size of the instruction window considered, and establish that there are many recurring reads and updates of the same register operands in most GPU computations. To exploit this opportunity, we propose Breathing Operand Windows (BOW), an enhanced GPU pipeline and operand collector organization that supports bypassing register file accesses and instead passes values directly between instructions within the same window. Our baseline design can only bypass register reads; we introduce an improved design capable of also bypassing unnecessary write operations to the RF. We introduce compiler optimizations to help guide the write-back destination of operands depending on whether they will be reused to further reduce the write traffic. To reduce the storage overhead, we analyze the occupancy of the bypass buffers and discover that we can significantly down size them without losing performance. BOW along with optimizations reduces dynamic energy consumption of the register file by 55% and increases the performance by 11%, with a modest overhead of 12KB increase in the size of the operand collectors (4% of the register file size).**

***Index Terms*—operand bypassing, GPU, register file, microarchitecture, compiler**

I. Introduction

Graphics Processing Units (GPUs) have emerged as an important computational platform for data-intensive applications in a plethora of application domains. They are commonly integrated in computing platforms at all scales, from mobile devices and embedded systems, to high-performance enterprise-level cloud servers. GPUs use a massively multi-threaded architecture that exploits fine-grained switching between executing groups of threads to hide the latency of data accesses. In order to support this fast context switching at scale, GPUs invest in large Register Files (RF) to allow each thread to maintain its context in hardware. The amount of parallelism available on a GPU (e.g., number of streaming multiprocessors, or SMs) has been steadily increasing as GPUs continue to grow in performance and size, which in turn increases the number of concurrent thread contexts needed to keep these units utilized [1]–[8].

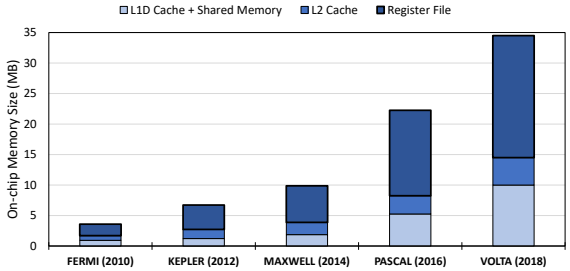


Fig. 1: On-chip memory components size in NVIDIA GPUs (from 2010–2018).

The large register file accounts for an increasingly larger fraction of on-chip storage, as shown in Figure 1. For example, in NVIDIA Pascal GPU, register file size is 14 MB, which accounts for around 63% of the on-chip storage area. Due to frequent accesses to the RF, it is a crucial microarchitectural component whose architecture substantially impacts the performance and energy-efficiency of GPUs. For example, port conflicts (in register file banks as well as operand collector units that collect the register operands) cause delays in issuing instructions as register values are read in preparation for execution. In addition, the RF has a large energy consumption footprint, since it is the largest SRAM structure that serves a large number of data accesses from the working threads. Earlier studies estimate that the register file is responsible for 18% of the total power consumption on a GPU chip [9], a percentage that has most likely increased as the size of RFs has continued to grow.

We propose a new GPU architecture technique, *Breathing Operand Windows (BOW)*, exploits the temporal locality of the register accesses to improve *both* the access latency and power consumption of the register file. More specifically, we observe that registers are often accessed multiple times in a short window of instructions, as values are incrementally computed or updated and subsequently used. As a result, a substantial fraction of register read and register write accesses can bypass the register file if mechanisms exist to forward them directly from one instruction to the next. This *operand bypassing* reduces dynamic access energy by eliminating regis-

978-1-7281-7383-2/20/$31.00 ©2020 IEEE
DOI 10.1109/MICRO50266.2020.00084

ter accesses (both reads and writes) from the RF, and improves overall performance by reducing port contention and other access delays to the register file banks.

BOW re-architects the GPU execution pipeline to take advantage of operand bypassing opportunities. Specifically, in the baseline design we consider operands reused within an instruction window: a key to increasing bypassing opportunities is to select the instruction window size carefully to capture register temporal reuse opportunities while maintaining acceptable overheads for the forwarding. To facilitate bypassing we dedicate an operand collector to each warp so that it can hold the set of active registers for that warp in a simple high performance buffering structure dedicated for each warp. Whenever a register operand is needed by an instruction, BOW first checks if the operand is already buffered so it can use it directly without the need to load it from the RF banks. If the operand is not present in the operand collector unit, a read request will be generated to the RF, which is sent to the arbitrator unit. In the baseline BOW, after an instruction finishes execution, the computed result is written back to both the operand collector unit as well as the register file (i.e., a write through configuration). This organization supports reuse of operand reads and avoids the need for an additional pathway to enable writing back values from the operand collector to the RF when they slide out of the window. Based on our experiments, BOW with a window size of 3 instructions reduces the physical register read accesses by 59% across all of our benchmarks. However, it does not support write bypassing since every write is still written to the RF; in fact, it increases the overhead for writes which are now written to both Operand Collector and RF.

In order to be able to capitalize on the opportunities for write bypassing, we introduce BOW-WR, an improved design that uses a write-back philosophy to overcome the redundant writes present in BOW. Specifically, the improved design writes any updated register values back to the operand collector only. When an instruction slides outside of the active bypass window its updated register value is written back to the RF *only if it has not been updated again by a subsequent instruction in the window* (in which case that first write has been bypassed since the update was transient). As described, BOW-WR shields the RF from some of the write traffic, but does not capture all write bypassing opportunities, and preserves some redundant and inefficient write behavior. Consider the following two cases: (1) Unnecessary OC writes: When a value will no longer be reused, writing it to the OC first, and then to the RF causes a redundant update. We are better off writing such value directly to the RF; (2) Unnecessary RF writes: When an updated register value is no longer live (i.e., it will not be read again before it is updated), it will be written back to the RF unnecessarily when the instruction slides out of the active window. In this case, we are better off not writing the value back to the RF.

Unfortunately, it is difficult to capture either of these opportunities directly in the architecture because they depend on the subsequent behavior of the program. Thus, to exploit the opportunity to eliminate these redundant write backs in BOW-WR, we task the compiler to do liveness analysis and classify each destination register to one of these three groups: those that will be written back only to the register file banks (to handle case 1 above); operands that will be written back only to the operand collectors (to handle case 2); and finally operands that first need to reside in operand collector and then due to their longer lifetime need to be written back to the register file banks for later use (this was the default behavior of BOW-WR before the compiler hints). We pass these compiler hints to the architecture by encoding the writeback policy for each instruction using two bits in the instruction. This compiler optimization not only substantially minimizes the amount of write accesses to the register file and fixes the redundant write-back issue, but also reduces the effective size of the register file as a significant portion of register operands are transient, not needed outside the instruction windows (52% with a window size of 3): we avoid allocating registers altogether in the RF for such values.

With respect to implementation, a primary cost incurred by the baseline BOW(and BOW-WR) is the cost of increasing the number of operand collectors (so that there is one dedicated per warp) as well as the size of each operand collector to enable it to hold the register values active in a window. With respect to increasing the number of OCs, we believe that this is in line with current trends in GPUs: While earlier Nvidia GPUs had a smaller number of operand collector units, starting from the Kepler series, the number of their operand collector units have increased. For example, NVIDIA TITAN X GPU (Pascal architecture) has 32 operand collectors which matches the maximum number of in-flight warps on an SM. With respect to the size of each OC, the baseline implementation adds additional entries to each operand collector to hold the operands within the active window (4 registers per instruction in the window). In the baseline implementation, this adds around 36KB of temporary storage for a window size of 3 across all OCs, which is significant (but still only around 14% of the RF size of modern GPUs). In order to reduce this overhead, we observe experimentally that this worst case sizing substantially exceeds the mean effective occupancy of the bypassing buffers. Thus, we provision BOW-WR with smaller buffering structures. However, since the available buffering can be exceeded under the worst case scenarios, we have to redesign the OCs to allow eviction of values when necessary. We also restrict the window size to the predetermined fixed window size and do not bypass instructions beyond the window size even if there is sufficient buffer space in the buffering structure. The reason behind this conservative choice is to facilitate the compiler analysis and tag the writeback target in BOW-WR correctly in the compiler taking into account the available buffer size. Without this simplifying assumption, an entry which is tagged by the compiler for no writeback to the RF may need to be saved if it is evicted before all its reuses happen. We are able to reduce the storage size by 50% with a performance reduction of less than 2% of the baseline BOW-WR. Considering other

overheads (such as modified interconnect), BOW requires an area increase of 0.17% of total on-chip area.

Because of the importance of the RF structure on GPUs, a number of prior studies have explored optimizations primarily to reduce its energy footprint. A number of works have explored different approaches to reduce the effective size of the register file [3], [10]–[12]. The effect of reducing the register file size is to improve the static energy consumption of the RF, but it does not impact the performance or the dynamic energy consumption of the RF. Most similar to our work, RF caching [13] adds a register file cache to keep the most commonly used data for each active warp, saving dynamic RF energy. This cache is organized like the original RF, but only smaller, and therefore there it improves energy but unlike BOW it does not improve performance.

We compare our work with register file caching and other related works in more detail in Section VI.

In summary, the paper makes the following contributions:

1) Introduces *Operand Bypassing*, a new technique in the context of GPU microarchitecture that capitalizes on the high temporal reuse of GPU register operands to substantially reduce accesses to the register file, improving performance and energy.
2) We leverage compiler liveness analysis to guide destination selection of the write-back register values, substantially reducing unnecessary write traffic. Bypassed transient values are also never allocated in the RF reducing the effective RF size.
3) We carry out occupancy analysis of the forwarding buffers and discover that their utilization is low. We propose to provisioning the operand collectors with smaller buffer structures to substantially reduce storage overhead.

Overall, BOW-WR improves IPC by 11%, and reduces dynamic energy of the RF by 55%, at a modest overhead of 0.17% increase in the total chip area, and 4% increase in storage (compared to the RF size). Section VII summarizes our conclusions and discusses potential future work.

II. Background

In this section, we overview the organization of modern GPU architecture, with a focus on the register file unit, to provide the necessary background for BOW. In the GPU execution model, a kernel is the unit of work issued typically from the CPU (or directly from another kernel if dynamic parallelism is supported). A kernel is a GPU application function, decomposed by the programmer into a grid of blocks mapped each to a portion of the computation applied to a corresponding portion of a typically large data in parallel. Specifically, the kernel is decomposed into Thread Blocks (TBs, also Cooperative Thread Arrays or CTAs), with each being assigned to process a portion of the data. These TBs are then mapped to streaming multiprocessors (SMs) for execution. The threads executing on an SM are then grouped together into *warps* (or *wavefronts* in AMD terminology) for the purposes of scheduling their issuance and execution. Warp instructions are selected and issued for execution by warp

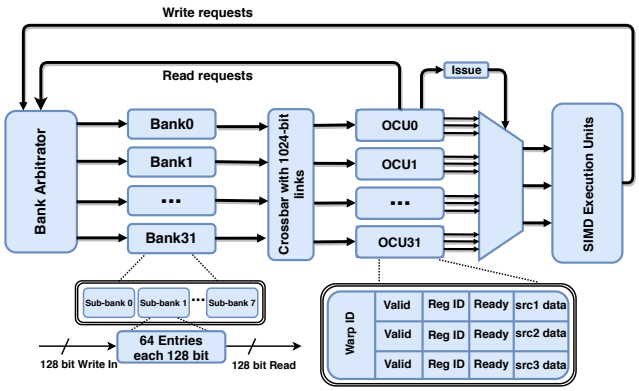


Fig. 2: Conventional GPU register file architecture (OCU0-OCU31 are Operand Collector Units).

schedulers in the SM (typically 2 or 4 schedulers, depending on the GPU generation). Warps that are assigned to the same warp scheduler compete for the issue bandwidth of that scheduler. In our baseline GPU (NVIDIA Titan X Pascal), there are four schedulers per SM, each able to issue two instructions per cycle to available GPU cores.

All the threads in a warp execute instructions in a lock-step manner (Single Instruction Multiple Thread, or SIMT model). Most GPU instructions use registers as their source and/or destination operands. Therefore, an instruction will access the Register File (RF) to load the source operands for all of its threads, and will write back any destination operand after the execution to the RF. The RF in each SM is typically organized into multiple single-ported register banks so as to support a large memory bandwidth without the cost and complexity of a large multi-ported structure. A banked design allows multiple concurrent operations, provided that they target different banks. When multiple operations target registers in the same bank, a *bank conflict* occurs and the operations are serialized, affecting performance.

Figure 2 shows the baseline register file organization for the Pascal generation of NVIDIA GPUs, with a size of 256 KB per SM split across 32 banks. A bank is made up of 8 sub-banks that are 128 bits wide each. All 32 registers belonging to the 32 threads in the same warp are statically allocated to consecutive sub-banks (in a single bank) with the same entry index. Thus, a full register for all the threads within a warp can be striped using one entry of one bank, allowing it to be operated on in a single cycle. Each bank can store up to 64 warp-registers.

When a warp instruction is issued for execution, an Operand Collector Unit (OCU) is assigned to it to collect its source operands values. Assuming 32-thread warps, each source operand (i.e., warp register) is 32 $thread \times 32\ bits = 128B$ in size. A warp's source operands are read from the RF banks and then buffered in the OCU. The operand collector units are not used to eliminate name dependencies through register renaming, but rather are used as a way to space register operand accesses out in time so that no more than one access

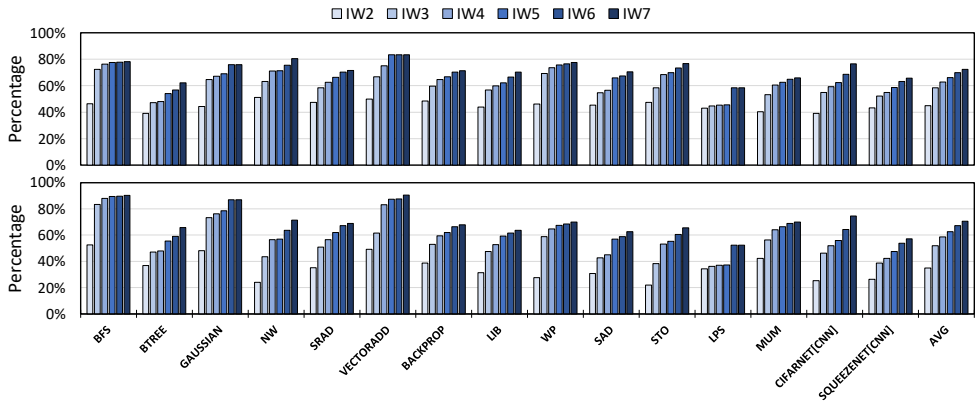


Fig. 3: Eliminated read (top) and write (bottom) requests through operand bypassing.

to a bank occurs in a single cycle. To reduce the interconnect network complexity, operand collectors are designed as single-ported buffers. An OCU fetches the register operands from the register banks they reside in, bound by the two following constraints: (1) *OCU port serialization*: Each OCU has only one port and therefore has to serialize reads when an instruction has multiple operands (NVIDIA GPU's use SASS whose instructions have up to 3 source operands); and (2) *Register bank conflicts*: While operands from different banks may be concurrently read from different OCUs, operands that access the same bank cause bank conflicts and cannot be issued together. The port constraints causing these conflicts are difficult to bypass by increasing the number of ports: the cost of a port is extremely high when considering the width of a warp register (128 Bytes).

Once all the source operands for a warp instruction are collected, it is ready for execution. Since each instruction may have up to three source operands [14], each OCU has three entries, each 128B to hold these operands. After the warp completes the execution, its results are written back to the RF, also competing for bank access with read operations. When this set of operations is performed repeatedly, it will generate many accesses to the large register file, and will incur a significant portion of the power consumed by the GPU. The RF also impacts performance due to the serialization that occurs due to port contention (in both register file banks as well as operand collector units).

III. Motivation

In this section, we motivate operand bypassing by studying register reuse patterns within different instruction window sizes. We use the GPGPU-Sim simulator [15], modeling a Pascal GPU. Given the in-order execution of GPUs, repeated accesses on operands within a small window of consecutive instructions are inevitable. Although we show results only for the Pascal architecture configuration, we repeated the results for Fermi and Volta configurations, which exhibit almost identical reuse statistics confirming that operand reuse patterns are computational properties rather than architecture dependent [16], [17]. Experiments in this paper use benchmarks from Rodinia [18], Parboil [19], NVIDIA CUDA SDK [14], and the Tango DNN Benchmark Suite [20].

Temporal locality in register operand accesses: In a conventional GPU Register File, each operand collector unit sends read requests for the source operands of one instruction (the one which currently resides in the operand collector unit). This read request process repeats independently for each individual instruction, with all register operands fetched from or written to the register file independently for each instruction. Our work is motivated by the observation that there is high temporal locality in the accesses of registers: in other words, the same register values are read and updated by nearby instructions, within a short window of instructions. If this is indeed the case, the traditional execution pattern where these operands are read and written repeatedly through the register file causes redundant expensive operations to the RF increasing both the power consumption of this large structure, as well as the access time due to the increased pressure on the limited ports of the RF.

To characterize the temporal reuse opportunity [21]–[23], we show in Figure 3 all bypassing opportunities for read (top) and write (bottom) requests to the register file, for different window instruction sizes and across 15 different benchmarks. An instruction window (IW) refers to a number of consecutive instructions from the same warp: an IW of 2 considers a sliding window of two instructions at a time and examines whether the operands of the first instruction are also needed by the second one. Note that a value that is reused in three consecutive instruction can continue to be bypassed even with an IW of 2 since the instruction window for bypassing is a sliding window. While we can bypass 45% of total read accesses and 35% of total write accesses to the register file with a window of just two instructions, a window of three instructions would eliminate substantially more accesses: 59% of total reads, and 52% of total writes on average. Beyond a window size of three instructions, the reuse opportunities continue to increase slowly, reaching over 70% with an instruction window of 7. Clearly, if we save this portion of register file accesses, we can substantially improve the dynamic energy consumption of the register file (by reducing the number of RF accesses) as well

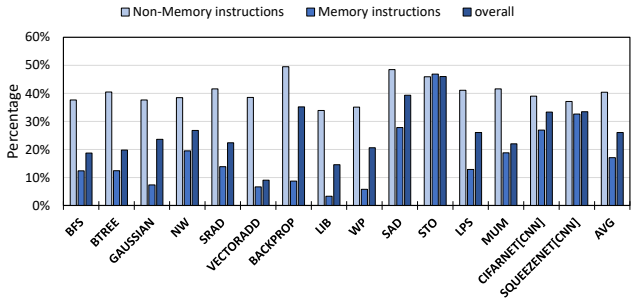


Fig. 4: Average time taken by operand collection stage for memory vs. non-memory instructions.

as performance (by reducing access time and port contentions in register file banks). A larger window size increases reuse opportunities, but comes at the price of wider (bigger) operand collectors which increase the area and energy consumption within those components. An effective BOW configuration balances these competing considerations.

Impact of operand collection stage latency on performance: The operand collection stage of the GPU pipeline holds issued instructions until their operands can be collected, typically from the register file. Figure 4 shows a breakdown of the percentage of cycles taken on average for memory instructions versus non-memory instructions within the operand collection stage of the pipeline. In our experiments, we excluded the amount of time spent for an instruction to be fetched; total execution time assumed to be from the moment that an instruction is scheduled by one of the warp schedulers until it finishes execution. Overall, about a quarter of the instruction execution time (and up to 47% for benchmarks such as STO) is spent in the operand collector unit. We note that this percentage is skewed by memory access instructions which have long execution times as well as a fewer number of operands (especially `global load` and `global store` instructions with cache misses). The operand collector unit consumes a substantial percentage of the execution time of non-memory instructions, as depicted in Figure 4. The primary delays in the OC occur while registers are read from the RF and are being collected in the single-ported operand collectors. As discussed previously, register reads for each OC are serialized since it is a single-ported buffer-like structure. Moreover, some reads are delayed due to register bank conflicts. With bypassing, as we decrease RF traffic, and with more operands already available in the OC, we expect the time spent in the OC to significantly decrease, improving overall performance.

IV. Breathing Operand Windows

In this section, we overview the design of BOW, the proposed architecture which exploits high temporal operand reuse to bypass having to read and write reused operands to the register file. We also introduce a number of compiler and microarchitectural optimizations to improve reuse opportunities, as well as to reduce overheads. BOW consists of 3 primary components. (1) Bypassing Operand Collector (BOC) augmented with storage for active register operands to

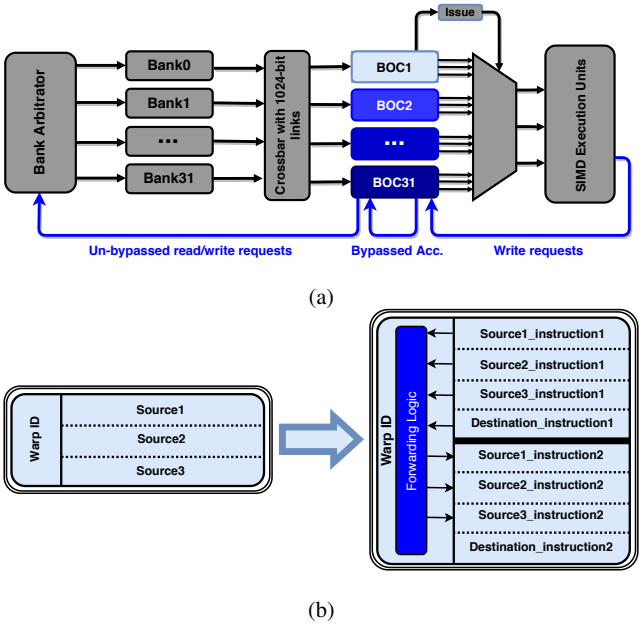


Fig. 5: (a) An overview of BOW. BOCX is Bypassing Operand Collector assigned to Warp X; (b) Baseline operand collector unit (left) compared to the proposed wider Bypassing Operand Collector (BOC) unit with forwarding logic support (right).

enable bypassing among instructions. Each BOC is dedicated to a single warp; this restriction simplifies buffering space management since each buffer is accessed only by a single warp. The sizing of the BOC is determined by the instruction window size within which bypassing is possible; (2) Modified operand collector logic that considers the available register operands and bypasses register reads for available operands (whereas baseline operand collectors fetch all operands from the RF); and (3) Modified write-back pathways and logic which enable directing values produced by the execution units or loaded from memory to the BOCs (to enable future data forwarding from one instruction to another) as well as to the register file (for further uses out of the current active window) in the baseline design. The writeback logic is further optimized with compiler-assisted hints in the improved BOW-WR.

A. BOW Architecture Overview

Figure 5 overviews the proposed architecture highlighting the primary changes and additions. The design centers around new operand collector unit additions, called the Bypassing Operand Collectors (BOC) in our design, that will allow the GPU to bypass RF accesses. Each BOC is assigned to a single warp (*BOC0-BOC31*) in Figure 5(a). While the operand collectors in our baseline architecture have three entries to hold the data of the source operands of a single instruction (Figure 5(b), left), BOW widens the operand collectors to enable the storage of source and destination register values for the usage of subsequent instructions (Figure 5(b), right). In addition, the forwarding logic in the BOC will check whether the requested operands are already in the BOC so will be

sent to the next instruction. Similar to the baseline architecture, and to avoid making the interconnection network more complicated, BOCs have a single port to receive operands coming from the register file banks. However, the forwarding logic within the BOCs allows forwarding multiple operands available in the forwarding buffers when an instruction is issued. In the baseline design, we conservatively reserve four entries per each instruction in the BOC to match the maximum possible number of operands which is three source operands plus one destination. Later we show that such conservative sizing is rarely needed, enabling us to provision the BOC with substantially smaller storage.

Instructions for the same warp are scheduled to the assigned BOC in program order as the instruction window slides through the instructions. When instruction `x` at the end of the window is inserted into the BOC, the Forwarding Logic checks if any of the required operands by instruction `x` is already available in the current window, then the oldest instruction (first instruction in the current window) with its operands are evicted from the window to make room for the next instruction, which will become available when the window moves. It is important to note that the instruction window is sliding; every time an operand is used by an instruction it remains active for window size instructions after that. If it is accessed again in this window, its presence in the BOC is extended in what we refer to as the *Extended Instruction Window.* When a branch occurs, the BOC waits until the next instruction is determined.

Instructions from *different* BOCs are issued to the execution units in a round-robin manner. As soon as all the source operands for an instruction are ready (which potentially have been forwarded directly within the active window and without sending read requests to the register file), the instruction is dispatched and sent to the execution unit. When the execution of an instruction ends, its computed result is written back to the assigned BOC (to be used later by next instructions in the window). In the baseline BOW, this value is also written back to the register file (for potential later uses, if any, by an instruction out of the current window). It is worth mentioning that only the pathway from execution units to the BOCs has been added in our design thusfar, as the pathway from execution units to the register file is already established in the baseline architecture. While such simple write-through policy minimizes the complexity, it suffers substantial of redundant write backs (to the BOCs as well as register file); an inefficiency which will be addressed in BOW-WR.

Please note that *two dependent instructions* (where there is a RAW or WAW dependency between them) can never be among the ready to issue instructions within the same BOC. The scoreboard logic checks for this kind of dependencies prior to issue instructions to the operand collection stage (this is actually done when a warp scheduler schedules an instruction). Having an instruction in one of the BOCs means that it has already passed the dependency checks and its register operands exist either in the BOC or the register file. For independent instructions, there is no delay for bypassing: both can start executing, and even finish out-of-order.

```
1  //write to $r3, immediate use in line 14
2  ld.global.u32 $r3, [$r8];
3  mov.u32 $r2, 0x00000ff4;
4  mul.wide.u16 $r1, $r0.lo, $r2.hi;
5  mad.wide.u16 $r1, $r0.hi, $r2.lo, $r1;
6  shl.u32 $r1, $r1, 0x00000010;
7  mad.wide.u16 $r0, $r0.lo, $r2.lo, $r1;
8  add.half.u32 $r0, s[0x0018], $r0;
9  add.half.u32 $r0, $r9, $r0;
10 add.u32 $r1, $r0, 0x000007f8;
11 ld.global.u32 $r2, [$r1];
12 Shl.u32 $r2, $r2, 0x00000100
13 Add.u32 $r4, $r2, 0x0000008f;
14 set.ne.s32.s32 $p0/$o127, $r3, $r1;
```

Fig. 6: Code snippet from BTREE application illustrating bypassing operation in BOW.

B. BOW-WR: Compiler-guided writeback

BOW exploits read bypassing opportunities, but is not able to bypass any of the possible write operations as every computed value is written not only to the RF, but also to the BOC, following a write-through policy for simplicity. However, write bypassing opportunities are important: often a value is updated repeatedly within a single window. For example, consider `$r1` being updated by the instructions in lines 4, 5, and 6 of Figure 6; it only needs to be updated in the RF after the final write.

BOW-WR approaches bypassing using a write-back philosophy to enable write bypassing. In the simplest cast, it writes the computed results always to the BOC to provide opportunities for both read and write bypassing. When an updated operand slides out of the current active window, the forwarding logic checks if it has been updated again by a subsequent instruction within the active window. If so, the write operation will be bypassed, allowing the consolidation of multiple writes happening within the same window. In our prior example (Figure 6), when instructions 4 and 5 slide out of the active window, their updated `$r1` is discarded since in each case `$r1` is updated again within the window. When instruction 6 slides out, the value is written back (since neither instruction 7 nor 8 update `$r1`). The primary cost of BOW-WR (write-back instead of write-through) is that a new pathway needs to be established from BOCs to the RF.

Although using a write-back philosophy [24] significantly reduces the amount of redundant writes to the register file (Table I), it is not able to bypass all such write operations; in many instances, as an operand slides out of an active window, it is written back from the BOC to the register file while it is not actually going to be used again by later instructions (the operand is no longer live). Another source of inefficiency arises since computed operands are always written back to the BOC; if these operands are not needed again in the active

window, they could have been written directly to the RF, eliminating the write to the BOC.

In either of these situations, unfortunately, the microarchitecture does not have sufficient information to identify the optimal target of the writeback, since it depends on the future behavior of the program which is generally not visible at the point where the writeback decisions are made, leading to the redundant writes. Thus, to enable elimination of these redundant writes, we rely on the compiler to analyze the program and guide with the selection of the write back target. Specifically, the compiler performs liveness analysis and dependency checks to determine if the output data from an instruction should be written back only to the register file bank (when it will not be used again in the instruction window), only to the bypassing operand collector (for transient values that will be consumed completely in the window and no longer live after it), or both (which is the default behavior without the compiler hint). When we avoid writing values back to the RF, we reduce the pressure on the RF and avoid the cost of unnecessary writes for operands that are still in use. Similarly, when we write data to the BOC which is not going to be used, we pay the extra cost of this write only to later have to save the value again to the RF. An interesting opportunity also occurs in that transient values that are produced and consumed completely within a window, no longer need to be allocated a register in the RF. We discover that many operands are transient, leading to a substantial opportunity to reduce the effective RF size. Compiler-guided optimizations will allow us to avoid unnecessary writes and minimize energy usage. Table I shows the needed number of write accesses to the RF for the code in Figure 6 in the different versions of BOW(note that BOW write-through is identical to the unmodified GPU).

Destination Operand	# of write accesses to the Register file in:		
	BOW (write-through)	BOW (write-back)	BOW-WR (compiler Opt.)
$r0	3	1	0
$r1	4	2	1
$r2	2	1	0
$r3	1	1	1
Total	10	5	2

TABLE I: Number of write operations to the register file for code snippet shown in Figure 6.

There are three possible actions which can be taken after an instruction's output value is generated, which we explain using a piece of code from BTREE kernel as shown in Figure 6. Please note that in the following explanation, we assume the *instruction window size* is 3 (each sliding window contains three consecutive instructions).

1) Reuse outside of instruction window: The first instruction in Figure 6 (`ld.global` in line 2) loads the data from the global memory into `$r3`. Reuse of `$r3` occurs in the `set.ne` instruction in line 14. Since the first use of `$r3` is outside of the instruction window (please recall that window size is 3), the compiler liveness analysis marks `$r3` to be written back directly to the register file as there is no bypassing opportunity within the window containing `ld.glonal` instruction.[1] In this case, where the first reuse distance is greater than the window size, there is no need to write this value back to the bypassing operand collector. Figure 7 shows the breakdown of the instruction writes into the three categories that we are in the process of explaining. The leftmost bar represents this case where reuse is outside the instruction window, and occurs on average, in 21% of the computed operands. In this case, writing these operands to the BOC is unecessary.

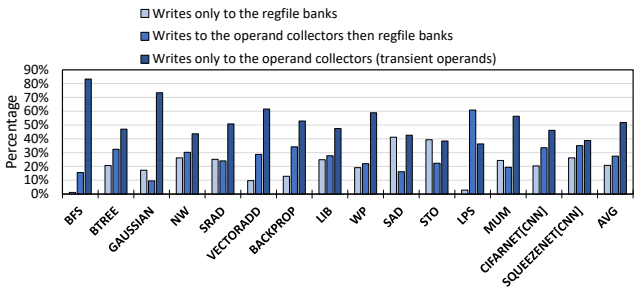


Fig. 7: Distribution of write destinations in BOW-WR

2) Reuse inside of instruction window: Operand reuse occurs in this case as a produced value is consumed again in the window. For example, `mov` instruction in line 3 of Figure 6 writes into `$r2`, where the immediate reuse of `$r2` happens in the next instruction (`mul` instruction in line 4). `$r2` will be used one more time in this window by the third (last) instruction of this window (`mad` in line 5). In this scenario, by writing `$r2` into the bypassing operand collector, we can directly forward it to the next instructions. Later on, the `mad` instruction in line 7 will also read `$r2`, which can be forwarded from previous `mad` instruction (in line 5) as they fall within the same window. As another example, the `add` instruction in line 10 writes into `$r1`. Later on, this register will be accessed by the next instruction (`ld.global` in line 11) as a source operand. We can directly forward the value from `add` to `ld.global` by storing it in the bypassing operand collector. However, this is not the end of lifetime for `$r1` since it will be used later in `set.ne` instruction in line 14 (please note that the value could not be forwarded from `ld.global` to `set.ne` as they do not belong to the same instruction window)

In the case where there is reuse of the value in the active window, we obviously would like to write it to the BOC to enable this reuse. However, these reusable cases break into two categories based on whether we will eventually need to save the register back to the RF or not (avoiding the write altogether). We describe these two cases next.

(a) Reuse of a transient operand: In some instances, a value produced by an instruction will be reused only while it resides in the bypassing operand collector. As a result, the value's lifetime does not exceed the instruction window, meaning that

[1]Further compiler optimizations to reorder instructions to increase bypassing opportunities are possible but we did not pursue this opportunity in the current version of our implementation.

there is no need for write backs to the register file bank once the instruction slides out of the instruction window. Lines 3, 4, 5, and 7 of Figure 6 show a case where an output value (register `$r2` in line 3) should be written back only to the BOC, because it is only used by future instructions *already within the same window* (or within the window of neighboring instructions). In this case, the write-back to the RF is bypassed since the `$r2` is a transient register value. In some cases, if this value came directly from memory or was produced by another instruction, we do not need to allocate a physical register in the register file for it. In this case, the immediate reuse distance across all the accesses is always less than IW. Figure 7 shows that these kinds of writes account for 52% of all operands. If indeed we can avoid allocating registers in many of these instances, we can further gain efficiency by reducing the effective size of the register file, allowing us for example to provision the GPU with smaller RF for the same performance, or gain performance by allowing additional Thread blocks for the same register file size.

(b) Reuse of a persistent operand: When a value is reused in the window, but continues to be live and will be reused later in the program, it must be saved back to the RF when it is evicted from the BOC. Lines 10, 11, and 14 of Figure 6 show such a case where the output value of the `add` (`$r1`) is going to be used within the same window (by the `ld.global` instruction), so it has to be written back to the corresponding BOC to take advantage of this bypassing opportunity. However, when the `add` operand is evicted from the BOC (to be replaced with the instructions in the next window), we need to write back `$r1` to the register file banks as well, as it is going to be used later by another instruction outside of the current window (which is `set.ne` in this example). More specifically, first read operation on register `$r1` will be simply bypassed within the BOC (as the value is being used immediately in the `ld.global` instruction). After this point, `$r1` is still alive but it is not going to be used within the same instruction window (distance is more than Instruction Window Size), so it has to be written back to the register file bank at the time of eviction for later use by the very bottom instruction. Figure 7 shows that 27% of all values fall in this category with window size of 3.

Identifying which of the three cases above to implement the most effective writeback option requires the ability to predict of how a register will be reused within and even outside the active register window. We elected to use compiler analysis to identify the type of each instruction writeback to provide hints to the architecture to identify the correct action for register writes. We use two additional flags in every instruction with a destination register, to indicate where the output data should be written. One bit is to enable writing to the BOCs, while the other bit is to enable writing back to the RF. Table I showcases the effectiveness of such compiler optimizations on the number of write operations on the register file. BOW with write-back policy is able to bypass a fraction of write operations. For example, the two consecutive writes into `$r1` in lines 4 and 5 of figure 6 could be bypassed as there `$r1` is being updated

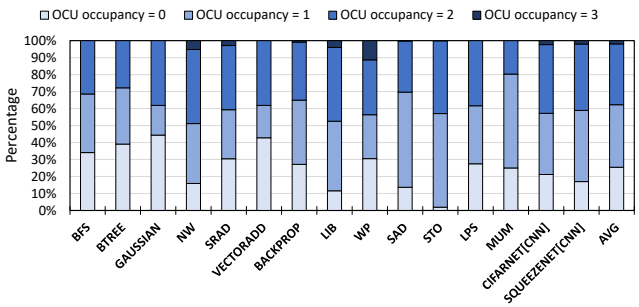


Fig. 8: Operand collector units' occupancy.

immediately by the next instruction. However, operands such as `$r2` in line 3 has to be written back to the register file as they slide out of the window, as there is no information on their future uses. Such redundant writes are avoided by the compiler annotations as `$r2` immediate reuse distance across all the accesses is always less than IW. Moreover, with the proposed compiler optimization, useless writes to the BOCs (about 21% of all write operations according to Figure 7) are avoided (for example, `$r3` in line 2 where it is not going to be used within the active window).

C. Reducing the Bypassing Storage Space

Thusfar, we have assumed that we provision each BOC conservatively with 4 registers for each instruction in an IW to account for the maximum possible storage required. The total size of a single bypassing operand collector will be $4 \times 128B \times 3$, or $1.5KB$ which is four times larger than an operand collector in our baseline architecture (which is $3 \times 128B = 384B$). However, we believe that these structures' occupancy likely to be low for the following three reasons.

- There are substantial bypassing opportunities within a window (nearly 60% with IW 3 as we saw in Figure 3). Only a single value of a reused register is stored and shared as it is forwarded to reusing insturctions.
- In the NVIDIA ISA, most instructions do not require three source registers. As shown in Figure 8, on average only 2% of the instructions need all three entries in the operand collector unit. For some applications such as BFS, BTREE, and LPS, no instructions with three register source operands are used. Please note that $OCU occupancy = 0$ corresponds to those instructions that do not have any *register source operand* (such as `NOP` and `RET` with no source operand, or `SSY` and `BRA` with immediate operands).
- With the compiler optimizations, a considerable fraction of computed values (about 21% of total write operations) are not written back to the BOCs as they have no reuse within the window. In those situations, we do not use the entry for the destination register in the BOC.

To confirm our intuition, we analyze the occupancy of the conservatively sized operand collector (four entries per each instruction) for a window of three instructions. As Figure 9 shows, about half the benchmarks (for example, BFS, BTREE,

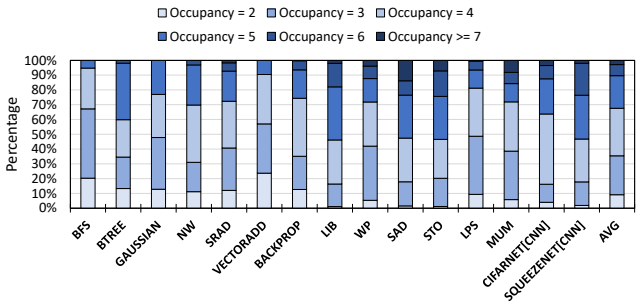


Fig. 9: BOC occupancy with a window 3: half of the entries are unused.

and BACKPROP) never need more than half of the entries in each BOC. Even benchmarks with higher occupancy (like WP and SAD), do not use many of the available registers. On average, across all of our benchmarks, only 3% of the cycles require more than 50% of the available entries. There were no instances where the worst case scenario (all 12 entries occupied) arose in our experiments.

Given this occupancy behavior, we reduce the buffer size in the BOC to half of the maximum possible size. As a result, there are situations where the occupancy is high and not all the operands within the window can be kept in the BOC, in which case we use a FIFO replacement policy to evict the oldest entry. We restrict the window size to the nominal window size (3 instructions in our example) and do not bypass instructions beyond the window size even if there is sufficient buffer space in the BOC. This allows us to simplify the compiler analysis and tag the writeback target in BOW-WR correctly in the compiler taking into account the available buffer size (without this simplifying assumption, an entry tagged for no writeback to the RF may need to be saved if it is evicted before all its reuse targets use it). In future work, we will consider enabling bypassing beyond the nominal window size limited only by the buffer space.

In worst case scenarios, sharing fewer number of entries in a BOC across multiple instructions may lead to an increase in the write-back traffic from BOC to the register file due to space limitation, which in turn can potentially hurt performance and energy efficiency of BOW-WR. However, given that this happens only 3% of the time across our benchmarks (Figure 9), this effect is small. At the same time, reducing the size of the buffering from 12 to 6 entries per BOC means that the storage overhead in the BOC was reduced from 4x to only 2x that of the baseline GPUs.

V. Evaluation

We use GPGPU-Sim [25] which models an NVIDIA TITAN X Pascal (GP102) configuration [26] shown in Table II. Benchmarks have been selected from the Rodinia [27], ISPASS [28], Parboil [19], and CUDA SDK [29] (see Table III).

A. Performance/Energy Evaluation

Performance: Figure 10 displays the normalized IPC improvement achieved by BOW and BOW-WR compared to the

# of SMs	56
# of cores per SM	128
Max # of TBs/Warps/Threads per SM	16/32/1024
Register File size per SM	256KB
L1 Cache/Shared Memory Size per SM	48KB/96KB
L2 Cache Size	3MB
Warp Scheduling Policy	GTO

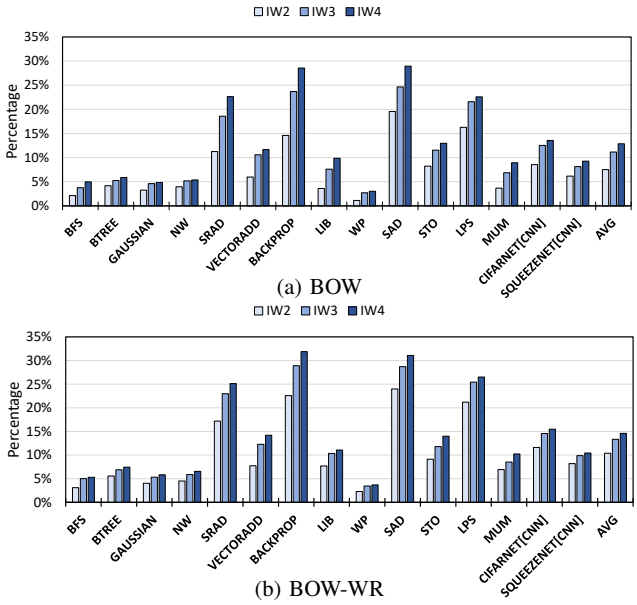
TABLE II: Nvidia TITAN X (Pascal Arch.) Configuration

Suite	Bench. Name	Description
ISPASS [28]	LIB	LIBOR Monte Carlo
	LPS	3D Laplace solver
	STO	StoreGPU
	WP	Weather prediction
Rodinia [27]	BackProp	Back-propogation
	BFS	Breadth first search
	BTree	Braided B+ Tree
	Gaussian	Gaussian elimination
	MUM	MummerGPU (Sequencing)
	NW	Needleman-Wunsch
	SRAD	Speckle Reducing Anisotropic Diffusion
Tango [20]	CifarNet	CifarNet NN
	SqueezeNet	SqueezeNet NN
CUDA SDK [29]	VectorAdd	Vector-Vector Addition
Parboil [19]	SAD	Sum of Absolute Differences

TABLE III: List of used benchmarks

baseline, using different instruction windows. As a result of bypassing substantial amount of read and write operations, port contention decreases (on both register file banks as well as BOCs), leading to better performance. Notably, we observe IPC improvement across all benchmarks. BOW-WR achieves marginally better performance due to its ability to reduce considerable amount of write operations, while BOW's improvement comes from bypassing the read operations. On average, with a window of three instructions, BOW and BOW-WR can improve the IPC by 11% and 13%, respectively. The small magnitude of advantage in IPC for BOW-WR over BOW is not surprising since writes are not on the critical path of instructions; however, as we will see later, the advantage of BOW-WR is higher in terms of energy savings. As IW grows beyond 3, we observe diminishing returns in the IPC improvements. Operand bypassing is more effective on register-sensitive applications (such as SAD). In contrast, benchmarks such as WP with lower register usage and fewer operand reuse opportunities gain little performance.

To evaluate the impact of reducing the storage size in the BOCs, we reduce the bypassing storage space by half (assuming window size of 3, each BOC has six entries instead of twelve, which are shared across every three consecutive instructions). Figure 11 shows the performance effect of this space optimization. While most of the benchmarks with lower BOC occupancy sustain their performance improvement under this space constraint, the IPC improvement slightly degraded for benchmarks with higher BOC occupancy such as SAD. On average, we have observed a 2% performance loss with half



(a) BOW

(b) BOW-WR

Fig. 10: IPC improvement.

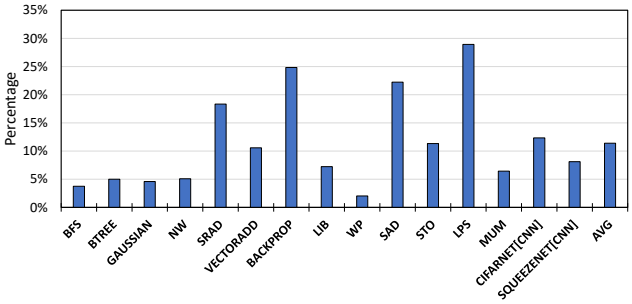


Fig. 11: IPC increase with 6-entry BOC (half-size).

bypassing storage; we still obtain nearly 11% IPC improvement even with half the storage size.

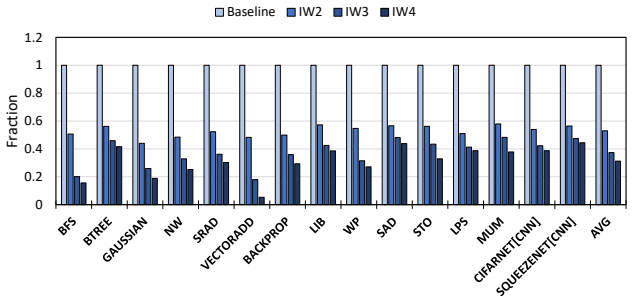
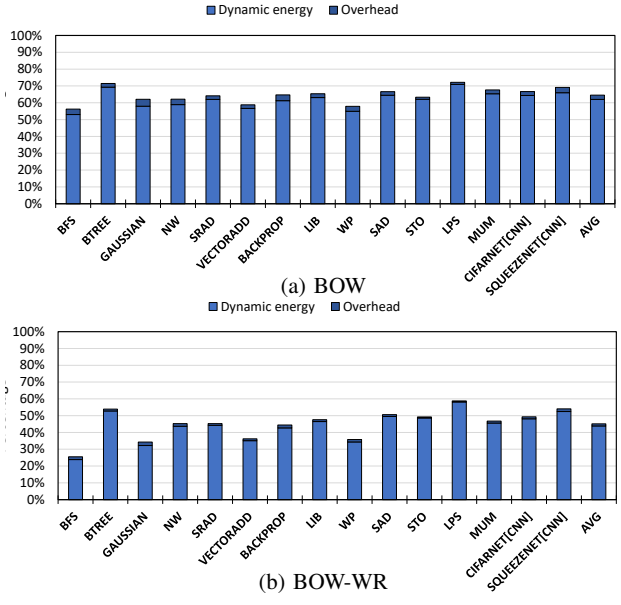


Fig. 12: Cycles spent in OC stage for different window sizes (normalized to the baseline).

Figure 12 shows the normalized time that each application spends in the operand collection stage. The OC residence time is reduced significantly when $IW = 2$ and 3 (by about 60% in the latter case). However, we do not see substantial additional benefit with larger windows. This result demonstrates that



(a) BOW

(b) BOW-WR

Fig. 13: Normalized RF dynamic energy

BOW is able to successfully find most of the reuse opportunities and reduce the OC residency in a consistent way for all applications. In most cases, the more time the application spends in the OC stage in the baseline case, the more benefit it can get from operand bypassing. However, we do not see that this hypothesis hold across all applications since the impact on IPC varies because the application performance may be bound by other stages of the pipeline.

RF Energy: Figure 13 shows the dynamic energy normalized to the baseline GPU for BOW and BOW-WR respectively. The small segments on top of each bar represent the overheads of the structures added by BOW. Dynamic energy savings in Figure 13 are due to the reduced number of accesses to the register file as BOW and BOW-WR shield the RF from unnecessary read and write operations. Specifically, BOWwith a window size of 3 instructions reduces RF dynamic energy consumption by 36%, after considering the 3% increase in overheads. BOW-WR is even more successful in saving dynamic energy because it also avoids substantial amount of write operations to the RF. We observed 55% reduction to the overall dynamic energy in BOW-WR, after considering 1.8% increase in overhead. Note that even the overhead of BOW-WR is lower since the eliminated writes also consume overhead through the additional BOW structures. The source of dynamic energy overhead include the accesses to the BOCs as well as the modified interconnect. It is interesting that although the IPC impact of BOWvaries across applications, the advantage in energy saving is relatively consistent: shielding the RF from operations reduces dynamic energy.

Comparison to Register File Cachine: Similar to BOW, there have been attempts to reduce the RF dynamic energy by caching the outputs in a smaller cache structure [13], a technique called register file cache (RFC). RFC was proposed

in conjunction with liveness information provided by the compiler and used by a two-level warp scheduler to reduce the energy consumption by 36%. Comparing RFC to BOW, there are at least two major differences:

- Register File Cache (RFC) is organized like the original RF, but only smaller and closer to the operand collectors. Hence, it does not resolve the port contention issues. In contrast, BOW resolves this by distributing the cache across operand collectors.
- In RFC, all computed results are written back to the cache, regardless of whether or not they will be used in near future. This leads to redundant cache write backs which are avoided in BOW-WR through compiler hints.

We implemented RFC with 6 entries per each thread (with 32 threads in each warp and 32 warps per SM), and on average, observed less than 2% performance improvement. Note that RFC overhead in this configuration is 24KB, which is double than that of BOW-WR with space optimization. BOW-WR also saves substantially more power by consolidate writes.

Hardware Overhead: The largest additional structure in BOW is the widened bypassing operand collectors. In baseline BOW with an $IW = 3$ configuration, each BOC holds 12 entries, each 128B wide (1.5KB in total), while in our baseline architecture, each operand collector is 384B. This adds around 36KB of temporary storage across all BOCs, which is significant (but still only around 14% of the total register file). However, we have showed that this choice is highly conservative. BOW-WR is able to sustain most of the performance improvement with half-sized buffer storage, which adds 12KB of temporary storage across all BOCs (4% of the total register file size). We calculate the BOC (in baseline BOW) and register file power consumption using CACTI v7.0 [30] and report them in Table IV.

Parameter	BOC	Register bank	Percentage
Size	1.5KB	64KB	2%
Vdd	0.96V	0.96V	-
Access energy	2.72pJ	185.26pJ	1.4%
Leakage power	1.11mW	111.84mW	0.9%

TABLE IV: BOC overheads in 28nm technology

To analyze the hardware overhead, we modeled a network consisting of the 32x32 crossbar, BOCs, bus arbiters, and the bus in RTL using Verilog and synthesized it in 28nm technology using Synopsys Design Compiler. The power and area of the register banks were modeled in CACTI. The design comfortably meets the timing constraint for 1GHz clock. The total energy for the redesigned BOC network is 33.2mW assuming a 50% of the cycles write. This is small compared to the 2.5W power of the whole register bank. The overall area of the added circuitry is less than $0.04mm^2$ compared to the $1.72mm^2$ size of a register bank: the additional network area is less than 3% of the area of a register bank, and less than 0.1% the area of the full RF.

VI. Related Work

Energy efficiency of GPUs has been an area of increasing importance [31]–[50]. These prior works have explored improving the performance or energy efficiency of GPU register files in a number of ways.

GPU Register File Scalability: Since the number of live registers in proportion to the total number of registers is relatively small, there are also trends to compress, reduce, and even replace the register file altogether so as to save power used for registers that would never need be utilized in most applications. Jeon et al [10] introduced a method to virtualize the RF addressing, which would leverage the variations in the instructions being executed by different warps, and allow the dead registers from one warp to be renamed and reallocated to another. As a result, this approach effectively shares the same physical registers with multiple warps during the course of a kernel run, with the performance suffering little to no loss with even the RF cut down to 50% of its size. However, it incurs a significant overhead. On a similar note, motivated by the low ratio of live registers for a CTA to its allocated registers, Oh et al [51] proposed FineReg, a new GPU architecture which would enable running more concurrent CTAs, increasing the RF utilization and the overall performance. It featured a RF divided into active and pending regions, with running CTAs using the active region, and stalled CTAs being moved to the pending region, allowing a new CTA to be run in the active region. RegMutex [2] improved performance by sharing a subset of physical registers between warps during the GPU kernel execution. However, it substantially increases the dynamic energy due to higher warp occupancy on the SM. RegLess [11] replaces the register file with a smaller staging unit with the help of compiler annotations, leveraging the short-lived and long-lived behaviors of the register. RegLess achieves lower power and smaller register storage size while maintaining performance. The Latency-Tolerant Register File (LTRF) [52] uses compiler-analysis to identify registers to move into a register cache, which enables tolerance of large register files. However, this higher performance comes at the cost of a larger, more power-hungry register file. Compared to these solutions, BOW is the only solution that improves both the energy consumption *and* the performance of the RF.

Register File Compression: Some of the state-of-the-art have targeted narrow register values, i.e. values which use less than half of their allocated register. Not only do they cause unnecessary energy leakage due to the storage of unused bits, but also there will also be additional delay when accessing those elements, since they are treated as a wide register when accessed, regardless of the actual value. With compiler optimizations, if narrow values are identified, two of them can be squeezed into a physical register, which leads to more RF utilization and less overall energy consumption. Wang et al [53] proposed a method to pack narrow values in the GPU register file in a greedy fashion, achieving an average speedup of 18%. On top of register packing, Esfeden et al [3] also proposed to coalesce register file accesses in order to reduce the number of RF accesses, thereby reducing dynamic energy consumption by 17% as well as improving the performance by coalescing multiple register read operations into a single physical access. Voitsechov et al [54] aimed to increase GPU

thread occupancy by identifying the last register usages within the code and releasing them at those points in the kernel for the usage of other threads. Combined with techniques for more efficient allocation for scalar and narrow values (de-duplication and packing, respectively), their method achieved an average of 12% speedup on register-bound workloads.

Related CPU register file optimizations: Register file efficiency has been an ongoing research topic in the CPU industry as well, having started long before the era of GPGPUs. Balkan et al [55] observed that a substantial fraction of computed values in a typical superscalar datapath are transient. They proposed a microarchitectural technique which predicts transient registers and avoids register allocations for predicted transient values. BOW also identifies and eliminates transient registers but does so using compiler hints. Park et al [56] predicted the bypassing opportunities to reduce the energy consumption, enabling register banks to be designed with lower number of ports. Swensen et al [57] addressed the issue that larger register sets have a longer access time than a smaller one, and based on this, proposed a hierarchical register set that contained close, middle and distant register sets, which would be used for different scenarios based on their time criticality, e.g. more critical tasks would be performed on the smaller, but closer, register set. There have also been other works that mainly target the access latency of CPU register sets [58]–[63]. The nature of RFs for superscalar CPUs is substantially different from those of GPUs, and therefore these techniques do not directly carry over to GPUs.

VII. Concluding Remarks

We observe that register values are reused repeatedly in close proximity in GPU workloads. We exploit this behavior to forward data directly among nearby instructions, thereby shielding the power-hungry and port-limited register file from many accesses (59% of accesses with an instruction window size of 3). Our best design (BOW-WR) can bypass both read and write operands, and leverages compiler hints to optimally select write-back operand target. BOW-WR reduces RF dynamic energy consumption by 55%, while at the same time increasing performance by 11%, with a modest overhead of 12KB of additional storage (4% of the RF size).

Acknowledgment

We would like to thank anonymous reviewers for their valuable feedback. This work is partially supported by the National Science Foundation under Grants CNS-1955650, CCF-2028714, CNS-1619322, and CCF-1815643.

References

[1] Nvidia, "Whitepaper: Nvidia's Next Generation CUDA Compute Architecture: Fermi", 2009.

[2] F. Khorasani, H. A. Esfeden, A. Farmahini-Farahani, N. Jayasena, and V. Sarkar, "Regmutex: Inter-warp gpu register time-sharing," in *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE Press, 2018, pp. 816–828.

[3] H. Asghari Esfeden, F. Khorasani, H. Jeon, D. Wong, and N. Abu-Ghazaleh, "Corf: Coalescing operand register file for gpus," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2019, pp. 701–714.

[4] Nvidia, "Whitepaper: Nvidia GeForce GTX 980", 2014.

[5] Nvidia, "Whitepaper: Nvidia's Next Generation CUDA Compute Architecture: KeplerGK110", 2012.

[6] F. Khorasani, H. A. Esfeden, N. Abu-Ghazaleh, and V. Sarkar, "In-register parameter caching for dynamic neural nets with virtual persistent processor specialization," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 377–389.

[7] H. Jeon, H. A. Esfeden, N. B. Abu-Ghazaleh, D. Wong, and S. Elango, "Locality-aware gpu register file," *IEEE Computer Architecture Letters*, vol. 18, no. 2, pp. 153–156, 2019.

[8] D. Tripathy, H. Zamani, D. Sahoo, L. N. Bhuyan, and M. Satpathy, "Slumber: static-power management for gpgpu register files," in *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*, 2020, pp. 109–114.

[9] 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 *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3. ACM, 2013, pp. 487–498.

[10] H. Jeon, G. S. Ravi, N. S. Kim, and M. Annavaram, "Gpu register file virtualization," in *Proceedings of the 48th International Symposium on Microarchitecture*. ACM, 2015, pp. 420–432.

[11] J. Kloosterman, J. Beaumont, D. A. Jamshidi, J. Bailey, T. Mudge, and S. Mahlke, "Regless: Just-in-time operand staging for gpus," in *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2017, pp. 151–164.

[12] S. Lee, K. Kim, G. Koo, H. Jeon, W. W. Ro, and M. Annavaram, "Warped-compression: enabling power efficient gpus through register compression," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3S, pp. 502–514, 2015.

[13] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron, "Energy-efficient mechanisms for managing thread context in throughput processors," in *2011 38th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2011, pp. 235–246.

[14] Nvidia, "Nvidia cuda sdk 2.3," [Online]. Available: http://developer.nvidia.com/cuda-toolkit-23-downloads, 2009.

[15] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 163–174.

[16] M. Bakhshalipour, M. Shakerinava, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Bingo spatial data prefetcher," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 399–411.

[17] M. Bakhshalipour, A. Faraji, S. A. V. Ghahani, F. Samandi, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Reducing writebacks through in-cache displacement," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 24, no. 2, pp. 1–21, 2019.

[18] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. Ieee, 2009, pp. 44–54.

[19] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," *Center for Reliable and High-Performance Computing*, vol. 127, 2012.

[20] A. Karki, C. P. Keshava, S. M. Shivakumar, J. Skow, G. M. Hegde, and H. Jeon, "Tango: A deep neural network benchmark suite for various accelerators," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE Press, 2019.

[21] M. Bakhshalipour, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Domino temporal data prefetcher," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 131–142.

[22] M. Bakhshalipour, S. Tabaeiaghdaei, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Evaluation of hardware data prefetchers on server processors," *ACM Computing Surveys (CSUR)*, vol. 52, no. 3, pp. 1–29, 2019.

[23] M. Bakhshalipour, P. Lotfi-Kamran, A. Mazloumi, F. Samandi, M. Naderan-Tahan, M. Modarressi, and H. Sarbazi-Azad, "Fast data

delivery for many-core processors," *IEEE Transactions on Computers*, vol. 67, no. 10, pp. 1416–1429, 2018.

[24] M. Kayaalp, K. N. Khasawneh, H. A. Esfeden, J. Elwell, N. Abu-Ghazaleh, D. Ponomarev, and A. Jaleel, "Ric: Relaxed inclusion caches for mitigating llc side-channel attacks," in *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2017, pp. 1–6.

[25] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 163–174.

[26] M. Khairy, J. Akshay, T. Aamodt, and T. G. Rogers, "Exploring modern gpu memory system design challenges through accurate modeling," *arXiv preprint arXiv:1810.07269*, 2018.

[27] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. Ieee, 2009, pp. 44–54.

[28] https://github.com/gpgpu-sim/ispass2009-benchmarks, 2009.

[29] NVIDIA, "Cuda toolkit," https://developer.nvidia.com/cuda-toolkit, 2007, accessed: 2018-04-11.

[30] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "Cacti 7: New tools for interconnect exploration in innovative off-chip memories," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 2, pp. 1–25, 2017.

[31] M. Abdel-Majeed and M. Annavaram, "Warped register file: A power efficient register file for gpgpus," in *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*. IEEE, 2013, pp. 412–423.

[32] M. Taram, A. Venkat, and D. Tullsen, "Mobilizing the micro-ops: Exploiting context sensitive decoding for security and energy efficiency," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 624–637.

[33] M. Sadrosadati, S. B. Ehsani, H. Falahati, R. Ausavarungnirun, A. Tavakkol, M. Abaee, L. Orosa, Y. Wang, H. Sarbazi-Azad, and O. Mutlu, "Itap: Idle-time-aware power management for gpu execution units," *ACM TACO*, 2018.

[34] A. Mirhosseini, M. Sadrosadati, B. Soltani, H. Sarbazi-Azad, and T. F. Wenisch, "Binochs: Bimodal network-on-chip for cpu-gpu heterogeneous systems," in *Proceedings of the Eleventh IEEE/ACM International Symposium on Networks-on-Chip*. ACM, 2017, p. 7.

[35] M. Abdel-Majeed, D. Wong, and M. Annavaram, "Warped gates: Gating aware scheduling and power gating for gpgpus," in *Microarchitecture (MICRO), 2013 46th Annual IEEE/ACM International Symposium on*, 2013.

[36] Q. Xu and M. Annavaram, "Pattern aware scheduling and power gating for gpgpus," in *Parallel Architectures and Compilation Techniques (PACT), 2014 23nd International Conference on*, 2014.

[37] D. Wong, N. S. Kim, and M. Annavaram, "Approximating warps with intra-warp operand value similarity," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[38] M. Abdel-Majeed, D. Wong, J. Kuang, and M. Annavaram, "Origami: Folding warps for energy efficient gpus," in *Proceedings of the 2016 International Conference on Supercomputing*, ser. ICS '16, 2016.

[39] O. Kayiran, A. Jog, A. Pattnaik, R. Ausavarungnirun, X. Tang, M. T. Kandemir, G. H. Loh, O. Mutlu, and C. R. Das, "μc-states: Fine-grained gpu datapath power management," in *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, ser. PACT '16, 2016.

[40] M. H. Santriaji and H. Hoffmann, "Grape: Minimizing energy for gpu applications with performance requirements," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[41] S. Mittal and J. S. Vetter, "A survey of methods for analyzing and improving gpu energy efficiency," *ACM Comput. Surv.*, vol. 47, no. 2, pp. 19:1–19:23, Aug. 2014.

[42] A. Sethia and S. Mahlke, "Equalizer: Dynamic tuning of gpu resources for efficient execution," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014.

[43] A. Sethia, G. Dasika, M. Samadi, and S. Mahlke, "Apogee: Adaptive prefetching on gpus for energy efficiency," in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, 2013.

[44] Z. Liu, S. Gilani, M. Annavaram, and N. S. Kim, "G-scalar: Cost-effective generalized scalar execution architecture for power-efficient gpus," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017.

[45] M. Abdel-Majeed, A. Shafaei, H. Jeon, M. Pedram, and M. Annavaram, "Pilot register file: Energy efficient partitioned register file for gpus," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017.

[46] A. Majumdar, L. Piga, I. Paul, J. L. Greathouse, W. Huang, and D. H. Albonesi, "Dynamic gpgpu power management using adaptive model predictive control," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017.

[47] G. Wu, J. L. Greathouse, A. Lyashevsky, N. Jayasena, and D. Chiou, "Gpgpu performance and power estimation using machine learning," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015.

[48] N. Chatterjee, M. O'Connor, D. Lee, D. R. Johnson, S. W. Keckler, M. Rhu, and W. J. Dally, "Architecting an energy-efficient dram system for gpus," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017.

[49] Z. Liu, D. Wong, and N. S. Kim, "Load-triggered warp approximation on gpu," in *Proceedings of the 2018 International Symposium on Low Power Electronics and Design*, ser. ISLPED '18, 2018.

[50] A. Abdolrashidi, D. Tripathy, M. E. Belviranli, L. N. Bhuyan, and D. Wong, "WIREFRAME: Supporting Data-dependent Parallelism through Dependency Graph Execution in GPUs ," in *MICRO '17: Proceedings of the 2017 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017.

[51] Y. Oh, M. K. Yoon, W. J. Song, and W. W. Ro, "Finereg: Fine-grained register file management for augmenting gpu throughput," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 364–376.

[52] M. Sadrosadati, A. Mirhosseini, S. B. Ehsani, H. Sarbazi-Azad, M. Drumond, B. Falsafi, R. Ausavarungnirun, and O. Mutlu, "Ltrf: Enabling high-capacity register files for gpus via hardware/software cooperative register prefetching," in *ACM SIGPLAN Notices*, vol. 53, no. 2. ACM, 2018, pp. 489–502.

[53] X. Wang and W. Zhang, "Gpu register packing: Dynamically exploiting narrow-width operands to improve performance," in *2017 IEEE Trustcom/BigDataSE/ICESS*. IEEE, 2017, pp. 745–752.

[54] D. Voitsechov, A. Zulfiqar, M. Stephenson, M. Gebhart, and S. W. Keckler, "Software-directed techniques for improved gpu register file utilization," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 15, no. 3, p. 38, 2018.

[55] D. Balkan, J. Sharkey, D. Ponomarev, and K. Ghose, "Spartan: speculative avoidance of register allocations to transient values for performance and energy efficiency," in *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, 2006, pp. 265–274.

[56] I. Park, M. D. Powell, and T. Vijaykumar, "Reducing register ports for higher speed and lower energy," in *35th Annual IEEE/ACM International Symposium on Microarchitecture, 2002.(MICRO-35). Proceedings.* IEEE, 2002, pp. 171–182.

[57] J. A. Swensen and Y. N. Patt, "Hierarchical registers for scientific computers," in *Proceedings of the 2nd international conference on Supercomputing*. ACM, 1988, pp. 346–354.

[58] H. Zeng and K. Ghose, "Register file caching for energy efficiency," in *Low Power Electronics and Design, 2006. ISLPED'06. Proceedings of the 2006 International Symposium on*. IEEE, 2006, pp. 244–249.

[59] R. Balasubramonian, S. Dwarkadas, and D. H. Albonesi, "Reducing the complexity of the register file in dynamic superscalar processors," in *Microarchitecture, 2001. MICRO-34. Proceedings. 34th ACM/IEEE International Symposium on*. IEEE, 2001, pp. 237–248.

[60] E. Borch, E. Tune, S. Manne, and J. Emer, "Loose loops sink chips," in *Proceedings Eighth International Symposium on High Performance Computer Architecture*. IEEE, 2002, pp. 299–310.

[61] J.-L. Cruz, A. González, M. Valero, and N. P. Topham, "Multiple-banked register file architectures," in *ACM SIGARCH Computer Architecture News*, vol. 28, no. 2. ACM, 2000, pp. 316–325.

[62] T. M. Jones, M. F. O'Boyle, J. Abella, A. González, and O. Ergin, "Energy-efficient register caching with compiler assistance," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 6, no. 4, p. 13, 2009.

[63] P. R. Nuth and W. J. Dally, "The named-state register file: Implementation and performance," in *High-Performance Computer Architecture, 1995. Proceedings., First IEEE Symposium on*. IEEE, 1995, pp. 4–13.