2018 IEEE International Symposium on High Performance Computer Architecture

WIR: Warp Instruction Reuse to Minimize Repeated Computations in GPUs

Keunsoo Kim and Won Woo Ro
School of Electrical and Electronic Engineering
Yonsei University, Seoul, Korea
{*keunsoo.kim, wro*}*@yonsei.ac.kr*

***Abstract*—Warp instructions with an identical arithmetic operation on same input values produce the identical computation results. This paper proposes warp instruction reuse to allow such repeated warp instructions to reuse previous computation results instead of actually executing the instructions. Bypassing register reading, functional unit, and register writing operations improves energy efficiency. This reuse technique is especially beneficial for GPUs since a GPU warp register is usually as wide as thousands of bits. In addition, we propose warp register reuse which allows identical warp register values to share a single physical register through register renaming. The register reuse technique enables to see if different logical warp registers have an identical value by only looking at their physical warp register IDs. Based on this observation, warp register reuse helps to perform all necessary operations for warp instruction reuse with register IDs, which is substantially more efficient than directly manipulating register values. Performance evaluation shows that 20.5% SM energy and 10.7% GPU energy can be saved by allowing 18.7% of warp instructions to reuse prior results.**

***Keywords*-GPU; microarchitecture; instruction reuse; register renaming**

I. Introduction

Modern GPUs can be viewed as multithreaded vector machines. In GPUs, a vector instruction is referred to as a warp instruction which executes tens of threads collectively. GPU vector units that execute warp instructions have tens of thread lanes [1], [2], [3], [4]. Thus, these large arrays of functional units consume substantial amount of energy. In this reason, previous works have focused on *spatial* optimizations to reduce energy per warp instruction in functional units [5], [6], [7], [8], [9], [10], [11], [12], [13] or in register files [14], [15], [16], [17], [18], [19], [20], [21].

However, observing from the *temporal* point of view, we noticed substantial amount of warp instructions are repeating warp computations performed closely in time, results in opportunities to bypass processing itself. A warp computation refers to a vector computation performed by a warp instruction, which is characterized as a combination of the opcode, immediate, input, and result values for all threads in a warp instruction. In our 34 benchmarks, 31.4% of dynamic warp instructions performed such repetitive warp computations.

In CPUs, the concept of instruction reuse [22], [23], [24] has been proposed to exploit such temporal redundancy. In the same spirit, this paper proposes to adopt the notion of instruction reuse in GPUs, namely warp instruction reuse. With instruction reuse, computation results are reused by future instructions to eliminate re-execution. Potential energy saving of reusing warp instructions is even more significant in GPUs compared to CPUs, because reuse enables to bypass computation for all of the threads in a warp instruction simultaneously. Reusing also saves energy for data movement, which is considered as another energy bottleneck [25].

Because our goal is to improve energy efficiency, the key challenge is how to efficiently perform the reuse operation. To reuse a previous result, a warp instruction must compare its input values with previous ones to identify they perform the same computation. However, operating on GPUs vectored warp operand values introduces excessive storage and energy overheads. For instance, a reuse design [22] needs a mechanism to replicate 1024-bit input and result values for every warp instruction execution.

We propose an efficient warp instruction reuse design for GPUs that minimizes manipulating on vector register values inspired from previous works [22], [23], [26], [24], [27]. Our technique builds on register renaming where a logical warp register can be remapped to a physical warp register as needed. The basic idea is to use physical register IDs as a proxy for register values. For instance, a 10-bit physical ID occupies less than 1% of a 1024-bit value.

As described in Figure 1, the proposed warp instruction reuse synergies with a register sharing technique called warp register reuse. At each instruction's completion, the logical destination is remapped to a physical register if that physical register already holds the result value. Eventually logical warp registers having the same value will be mapped to share one physical warp register.

We noticed warp register reuse enables instruction reuse without comparing register values since physical register IDs can represent the values in the logical registers. To determine if there is a previously computed result, we test the identity of input values between the issued and recorded instructions by comparing the physical register IDs of source operands, instead of comparing the operand values directly. Reusing becomes simple as the logical destination of the reusing instruction can be remapped to the physical destination of the reused instruction. We will discuss microarchitectural techniques to efficiently perform the reuse operations.

We also extend the proposed warp instruction reuse technique further to benefit load instructions. Allowing load instructions to reuse previously loaded data dramatically reduces the number of L1 cache or even memory accesses. For this load reuse, we developed a technique to prevent memory data hazards between loads and stores which is necessary to maintain memory consistency.

The performance of the proposed technique is evaluated using 34 GPU applications. The results show that 18.7% of warp instructions reused prior computation results and avoided re-executions. The energy reduction due to bypassed backend execution is 20.5% per SM, which turns into 10.7% reduction of GPU-wide energy consumption.

II. Baseline GPU Architecture

In this paper, we will use the GPU terminology and a GPU architecture inspired from NVIDIA GPUs [1], [2], [3], [4]. A GPU chip has multiple streaming multiprocessors (SM). SMs collectively process a warp of 32 threads. A register access on a warp processes 32 thread registers associated with the warp. For simplicity, we will use the term *instruction* often interchangeably with *warp instruction*, and *register* with *warp register*, unless otherwise mentioned.

A SM can interleave up to 48 concurrent warps divided into two groups of 24 warps each. At a cycle up to one warp instruction can be fetched from each warp group. Decoded instructions are stored in a per-warp instruction buffer. At a cycle the warp scheduler associated with each group selects one warp instruction to be issued to the execution pipeline. A scoreboard associated with each warp tracks register dependencies in the warp. The destination registers of an issued instruction are registered to the scoreboard to denote they are write-pending. The write-pending registers are compared with the operand registers of the next instruction to check if there is no hazard. Retired instructions remove their destination registers from the scoreboards.

SM backend consists of a register file and functional units. Each 1024-bit warp register access is processed by a group of 8 128-bit register banks operate in lockstep. A register bank has 1 read and 1 write ports, therefore each bank group can serve 1 1024-bit read and 1 1024-bit write at a cycle. The register file has 8 bank groups. An instruction is dispatched to the functional units (FUs) when all operands are read. There are four execution pipelines; two SP pipelines for integer and floating point operations, one SFU pipeline for special functions such as exponentials, and one memory pipeline. Each execution pipeline has 32 functional units to serve one warp instruction per cycle.

III. Motivation

A. Reuse Opportunities for Warp Instructions

We first quantify how frequently warp instructions repeat an identical warp computation performed closely in time. We used a sampling method to focus on only the recent computations that can be reused by other instructions soon. At every 1K warp instruction execution, we count the number of identical warp computations in the past 1K instructions. Control flow instructions (such as branches and barriers) and memory stores are always counted as not repeated regardless their inputs. The global average is obtained by averaging the results from all profiling windows in each benchmark execution.

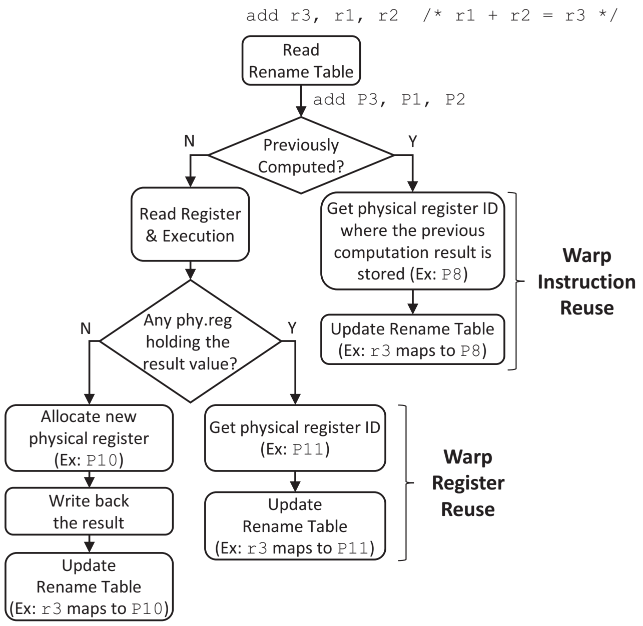

add r3, r1, r2 /* r1 + r2 = r3 */
Read Rename Table
add P3, P1, P2
Previously Computed?
N
Y
Read Register & Execution
Get physical register ID where the previous computation result is stored (Ex: P8)
Update Rename Table (Ex: r3 maps to P8)
Warp Instruction Reuse
Any phy.reg holding the result value?
N
Y
Allocate new physical register (Ex: P10)
Write back the result
Update Rename Table (Ex: r3 maps to P10)
Get physical register ID (Ex: P11)
Update Rename Table (Ex: r3 maps to P11)
Warp Register Reuse

Figure 1: Processing flow of issued instructions in the proposed backend pipeline.

Figure 2 presents the percentage of repeated computations. Across the 34 benchmarks 31.4% computations are repeated. We also observed 16.0% of computations are repeatedly appeared more than 10 times. For a computation repeated by 10 times, its computational cost can be reduced down to 1/10, because once it is first executed other 9 instructions can reuse the first result.

B. Sources of Repeated Warp Computations

Repeated warp computations appear in typical GPU coding patterns. Figure 3 is a kernel performing a filtering operation on an input image. We first focus on the computation for evaluating the highlighted sub-expression at line 15. The result of the sub-expression is uniquely determined from the *threadIdx* values. However, *threadIdx* is not uniquely assigned to each thread as threads are organized hierarchically. When threads belong to different thread blocks, an identical *threadIdx* value can be allocated to different threads. These threads will perform the identical computation as an identical input value is given to the sub-expression.

In the above example we can identify repeated computations directly from the code. However, certain repetitions

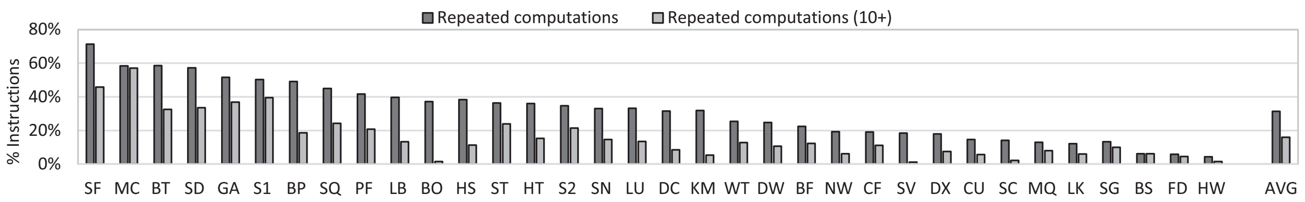


Figure 2: Percentage of repeated computations sampled for every 1K dynamic instructions.

```
__device__
ComputeSobel (const fScale, ul, um, ur, ml, mm, ...) {
  Horz = ur + 2*mr + lr - ul - 2*ml - ll;
  Vert = ul + 2*um + ur - ll - 2*lm - lr;
  Sum = fScale*(abs(Horz)+abs(Vert));
  ...
  return Sum;
}

__global__
SobelShared(const SharedPitch, ...) {
  __shared__ LocalBlock [...];
  SharedIdx = threadIdx.y * SharedPitch;
  ...
  for ( ib = threadIdx.x; ib < BlockWidth; ib += blockDim.x ) {
    pix00 = LocalBlock [SharedIdx+4*ib+0*SharedPitch+0];
    pix01 = LocalBlock [SharedIdx+4*ib+0*SharedPitch+1];
    pix02 = LocalBlock [SharedIdx+4*ib+0*SharedPitch+2];
    ...
    out.x = ComputeSobel(pix00, pix01, pix02,
                         pix10, pix11, pix12,
                         pix20, pix21, pix22, ... );
  }
  ...
}
```

Figure 3: An example kernel from the SF benchmark.

cannot be identified in this way, especially ones that appear according to the kernel's input. A call to *ComputeSobel* computes a filter output on a 3x3 region on the input image. If the function is called on different regions and all pixels in those regions have an identical pixel value, then the function repeats the identical computation although exactly the same input is given.

However, identifying which computations are repeated at compile time is inherently a challenging task, as the characteristics of the input given to individual instruction should be considered. Programmers can adopt a reuse technique at function level, such as in *ComputeSobel*, by using a lookup table that caches computation results for each input value given to the function. However, that increases complexity as an additional lookup is required for every call to the function. In this example, the function is relatively short, thus the increased complexity may not be justified.

IV. Warp Instruction Reuse

As explained earlier, it is quite frequent to see repeated computations in GPU applications. Therefore, reusing previous results can contribute to improving GPU energy efficiency. This section discusses a high-level operation of the proposed warp instruction reuse technique.

A. Conceptual Illustration

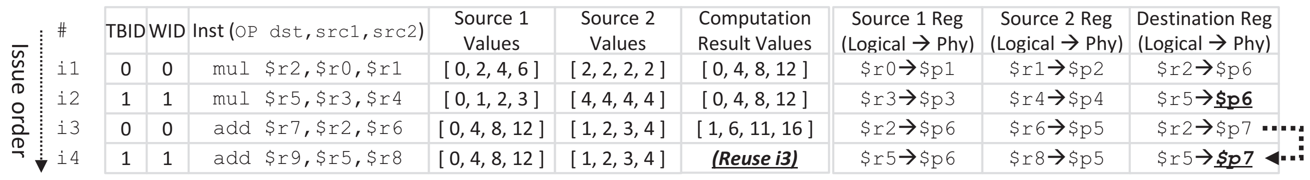
Figure 4 shows an example of warp execution stream. For simplicity, we assumed the warp size of 4 in this example. Each element in a vector `[]` denotes the operand value in each of the 4 thread lanes. Before executing a warp instruction, previously executed instructions are searched if there is any match of the opcode and input value. In this example, *i4* reuses the previous computation result performed by *i3*. We will refer to this execution simply as *i4* reuses *i3*. For arithmetic instructions reuses are allowed even when the previous instruction belongs to different warp or thread block, because as long as the operation is arithmetic the result is identical with the identical input.

B. Efficient Warp Instruction Reuse

For the reuse execution above, every warp instruction needs to record its opcode, input, and result values for future reuse. However, if we replicate the values in a separate buffer, both the required storage to store the values and the increased energy for communicating with the buffer may be prohibitively large because each register value is a 1024-bit wide vector.

We propose a more efficient reuse design that does not replicate register values. Our technique requires the ability to remap logical warp registers to physical warp registers dynamically. Basically, we remap logical warp registers having the identical architecturally visible value to point the same physical warp register. In Figure 4, initially four source registers of *i1* and *i2* are mapped to different physical registers `$p1` to `$p4`. When retiring *i2*, the logical destination `$r5` is remapped to `$p6` as the result value is already stored in `$p5`. In the same principle, we assumed previous executions have mapped `$r6` and `$r8` to the identical physical register `$p5`. Eventually, this technique prevents storing multiple copies of a warp register value in multiple physical registers. We refer to this register allocation technique as warp register reuse.

With warp register reuse, we can test identity of the input values by comparing just source physical register IDs. Although their logical IDs are different, we see the source registers of *i3* and *i4* have the identical physical



#	TBID	WID	Inst (`OP dst,src1,src2`)	Source 1 Values	Source 2 Values	Computation Result Values	Source 1 Reg (Logical → Phy)	Source 2 Reg (Logical → Phy)	Destination Reg (Logical → Phy)
i1	0	0	`mul $r2,$r0,$r1`	[0, 2, 4, 6]	[2, 2, 2, 2]	[0, 4, 8, 12]	`$r0`→`$p1`	`$r1`→`$p2`	`$r2`→`$p6`
i2	1	1	`mul $r5,$r3,$r4`	[0, 1, 2, 3]	[4, 4, 4, 4]	[0, 4, 8, 12]	`$r3`→`$p3`	`$r4`→`$p4`	`$r5`→**$p6**
i3	0	0	`add $r7,$r2,$r6`	[0, 4, 8, 12]	[1, 2, 3, 4]	[1, 6, 11, 16]	`$r2`→`$p6`	`$r6`→`$p5`	`$r2`→`$p7`
i4	1	1	`add $r9,$r5,$r8`	[0, 4, 8, 12]	[1, 2, 3, 4]	***(Reuse i3)***	`$r5`→`$p6`	`$r8`→`$p5`	`$r5`→***$p7***

Issue order

Figure 4: An example warp instruction reuse execution.

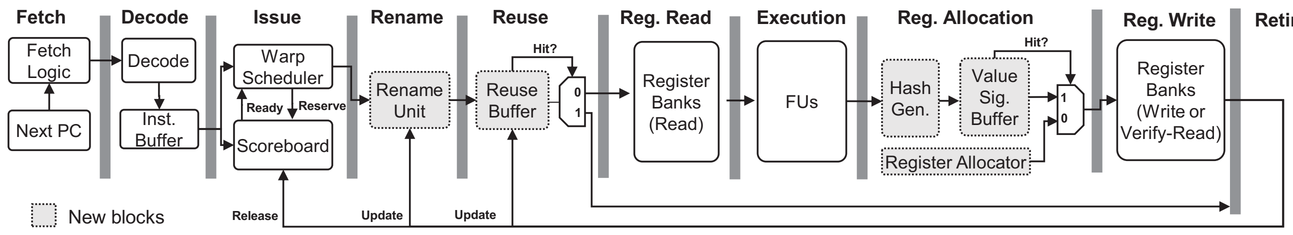


Figure 5: Pipeline of the proposed microarchitecture.

IDs thus their input values are identical. In addition to the testing, reusing itself becomes efficient. Instead of copying the reused instruction's result value to the new instruction's destination register, the new instruction can remap its logical destination to the reused instruction's result physical register. In this example, to reuse *i3*, *i4* remaps `$p7` to be its logical destination.

For this operation, we need to bookkeep each instruction's source and destination physical IDs, but we do not need to replicate or compare register values. Because each register ID is tens of bits wide, this design is significantly more efficient than using 1024-bit values directly. Another benefit of using register IDs is that instructions do not need to read the operand values from the register file for the identity test, thus we can also bypass register access for reused instructions. This advantage is substantial since register access in GPUs often consume more energy than computation [25], [14], [20].

V. Microarchitecture

Figure 5 shows a modified GPU pipeline with three new stages for warp instruction reuse. Each subsection discusses detailed operations in each new stages.

A. Register Allocation Stage

We will begin our discussion from the register allocation stage as other stages rely on how logical registers are mapped to physical registers. Given a result value, this stage determines whether we can remap the logical destination to the physical register that is already holding the value and bypass the writing operation. Figure 6 illustrates the flow of register allocation. When an instruction completes, the hash generation unit calculates the 32-bit hash from the 1024-bit result value. Then, the value signature buffer (VSB) uses the hash as a signature for values stored in physical registers. VSB searches its [hash, register ID] mappings to find an

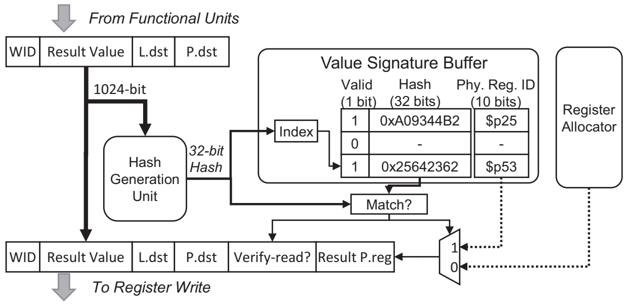


Figure 6: Operation in the register allocation stage. *L* and *P* indicate logical and physical IDs for source (*src*) and destination (*dst*) registers.

entry whose hash is identical to the result's hash. In our VSB design, we directly index an entry using the lower hash bits to avoid complexity for searching all entries associatively.

If there is no matched entry, a new physical register is allocated and the new hash-register tuple is registered to VSB. Once a register is registered to VSB, it can be shared by multiple logical registers. We allocate a new physical register for every VSB miss.

For a matched entry, VSB returns the physical register ID. However, this register's value can be different from the result value because a false positive match can happen due to a hash collision. Although false positives are very rare in our 32-bit hash space, a verification step is necessary to ensure correctness. Our solution is to read the register value and compare it with the result value, which we call a *verify-read* operation. As illustrated in Figure 7, if the register's value mismatches with the result value, then a new physical register is allocated and the bank arbiter schedules a write which stores the result value to the new register.

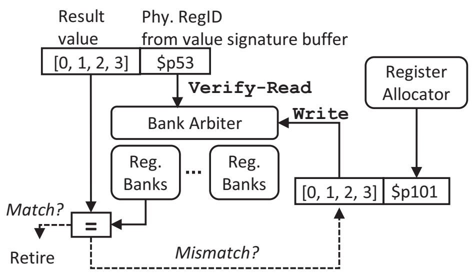


Figure 7: Verify-read operation in the register file.

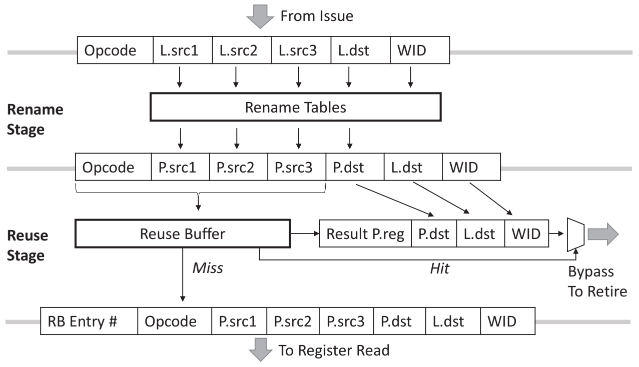


Figure 8: Information flow in rename and reuse stages.

B. Instruction Retire, Issue, and Rename

After an instruction passes a register verify-read or has written the value to a new register, the destination's new logical-physical mapping is sent to the rename unit where warp registers' logical-physical mappings are stored in rename tables. Each warp has a dedicated rename table. For instance, our baseline SM uses 48 rename tables for 48 concurrent warps in a SM. Each rename table has 63-entries to keep mapping for 63 logical registers in a warp. A table entry consists of a 10-bit physical register ID, a valid bit, and a pin bit. The pin bit is for handling branch divergence and will be further discussed in Section V-D. All entries are invalid at warp initialization, and the new mapping is written to the entry when a warp instruction retires.

After an instruction is issued, the physical register IDs for logical register operands are obtained from the rename tables. The upper half of the Figure 8 shows the relationship between logical and physical IDs in the rename stage.

It is worth noting that we use logical IDs for instruction issue and scoreboard operation. As in the baseline GPU, the scoreboard tracks register dependencies using logical IDs. The logical destinations are carried through the pipeline stages as required at retire.

C. Reuse Stage

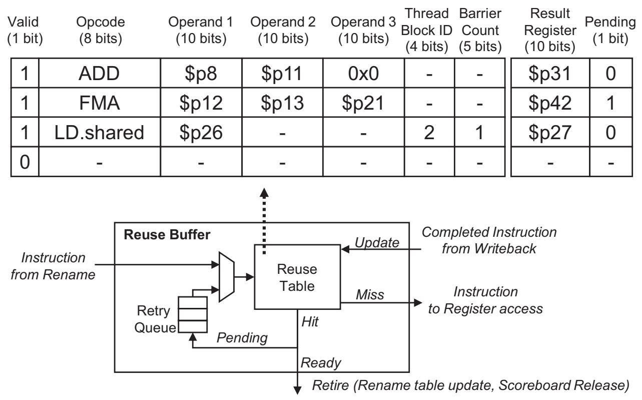
Renamed instructions are moved to the reuse stage to examine if there is any recorded reusable result. Figure 8 shows the flow of information in the rename and reuse stages. A renamed instruction's opcode and source register IDs are used as tag for looking up the reuse buffer. The reuse buffer is a cache-like structure that uses [opcode, physical register IDs of source operands] as tag, as shown in Figure 9. The operand fields can be used to store immediate values according to instruction type. As similar to the VSB case, the reuse buffer can be designed to associatively search all entries. However, we observed the benefit was marginal, and we chose the buffer to be directly indexed. The index is obtained by hashing the tag, and it is carried with the instruction for updating the buffer at retire.

For a renamed instruction, the buffer searches if there is an entry with the identical tag. When a matched entry is found, then the instruction is bypassed to the retire stage using the obtained result register ID where the reused result value is stored. Otherwise, the instruction continues to the register read stage for execution.

D. Handling Branch Divergence

A warp instruction is divergent when any of the 32 thread lanes are inactive during diverged control flow execution. In convergent instructions, remapping a logical warp register to a new physical warp register is relatively simple, because all lanes generate new values thus all thread registers can be filled completely. However, in divergent instructions only active lanes generate new values, thus the thread register values of inactive lanes must be copied from the current warp register to the new warp register. If divergent instructions require such copying operation for every new allocation, the overhead may be significant.

This issue can be handled more efficiently. When an instruction redefines a logical register for the first time in a diverged control flow, we allocate a new physical register dedicated to the logical register. The dedicated register can be overwritten safely if we ensure it is mapped only to that logical register. Thus, until reaching the end of the divergent flow, subsequent instructions can avoid copying overhead for the logical register by overwriting the dedicated register.



Valid (1 bit)	Opcode (8 bits)	Operand 1 (10 bits)	Operand 2 (10 bits)	Operand 3 (10 bits)	Thread Block ID (4 bits)	Barrier Count (5 bits)	Result Register (10 bits)	Pending (1 bit)
1	ADD	$p8	$p11	0x0	-	-	$p31	0
1	FMA	$p12	$p13	$p21	-	-	$p42	1
1	LD.shared	$p26	-	-	2	1	$p27	0
0	-	-	-	-	-	-	-	-

Reuse Buffer
Instruction from Rename
Retry Queue
Reuse Table
Update
Completed Instruction from Writeback
Miss
Instruction to Register access
Hit
Pending
Ready
Retire (Rename table update, Scoreboard Release)

Figure 9: Reuse buffer operation for warp instruction reuse.

For this purpose, we use a 1-bit flag called pin bit associated with each logical register. Each pin bit indicates if that logical register is currently mapped to a dedicated physical register. Initially all pin bits are cleared. The pin bit state is obtained when an instruction is renamed and carried to the register allocation stage. If the instruction is divergent and the destination's pin bit is unset, a new physical register is allocated and the pin bit is set. Because the instruction is divergent, the new register is not registered to VSB, which ensures the new register is uniquely mapped to the logical destination. The pin bit is cleared by the first convergent instruction that redefines the logical register.

When a new dedicated physical register is allocated, thread register values of inactive lanes are copied from the current physical register by a dummy MOV injected to the register file [20]. The dummy's active mask is obtained by flipping the current active mask. The source and destination register of the dummy MOV are mapped to the current and the new physical register, respectively. We observed dummy MOVs increase only less than 2% of the total instruction count, as they are injected only for the first writes to each logical register in a diverged flow.

Divergent instructions also do not reuse prior results. Because a reuse is a rename operation on a warp register, we may need to rename individual thread registers to reuse results only for a fraction of lanes, which is significantly more complex. Instead, divergent instructions simply bypass reuse buffer access at the reuse stage, and also do not update the reuse buffer at retire.

E. Register Management

The proposed mechanisms require the ability to dynamically allocate and release warp registers. The allocation is performed by a free register pool which has a list of free warp registers. The release is performed by a register reference counting system [23], [28], [29], [30]. A warp register is returned to the free pool when no reference to the register is left in rename tables, reuse buffer, or value signature buffer. Tracking registers' reference counts can be performed in parallel with instructions' execution, because instructions can continue execution as long as new registers can be allocated from the free pool.

We propose a reference counting system consists of a scheduler and a counter array. Each counter is associated with a physical register, and records how many references to the register are used in the tables and buffers. When the counter becomes zero the associated register is returned to the free pool. Since the counter must be updated whenever a register ID is registered or removed from the tables, the system has concurrent inputs as many as the register ID updates possible in a cycle. The scheduler merges multiple requests to the same counter. The system is pipelined to minimize impacts on clock frequency at a cost of latency from a table update until the register is released to the pool. However, this delay causes stalls rarely, because GPU applications tend to underutilize registers thus usually there are sufficient number of free registers available in the pool.

We considered two policies regarding how many physical registers should be allowed for reuse. First, *max-register* uses all available registers for maximizing reuse opportunity. For this policy, we must prevent a deadlock that can occur when no free register is left when a new register is requested. When no free register is left, the pipeline is switched to low register mode where a miss in the reuse buffer or the value signature buffer evicts the entry, or an entry is randomly evicted if there was no access in a cycle. Eventually the pipeline will be released from deadlock as registers are gradually released from the buffers.

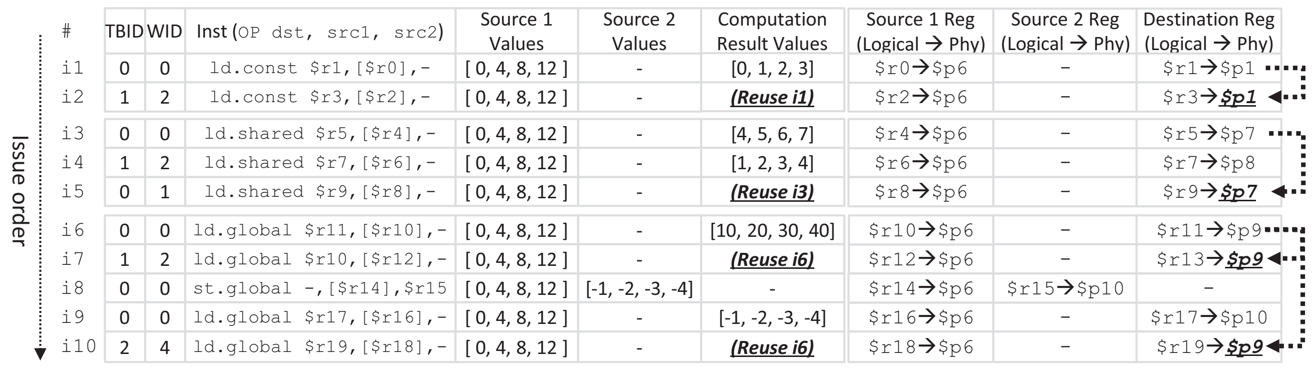
Another policy is *capped-register* where physical registers can be used up to the total logical register count, which is the number of logical registers used in a warp multiplied by active warp count. On a GPU where unused warp registers can be power-gated [17], [20], [31], this policy prevents possible increase in leakage energy to turn on additional registers for reuse. When the utilization reaches the limit, the pipeline is switched to the low register mode to keep the usage below the limit. We included both policies in our evaluation and observed significant energy saving regardless of the policy selection.

VI. Optimizations

A. Reusing Load Instructions

We have discussed reusing of arithmetic warp instructions, but this scheme can be further extended to reuse load instructions. A load can be viewed as repeating any previous load if both loads access the identical address and there is no store inbetween. Usually this repeated fetching may be served by the L1 cache. Instead, a load can directly reuse a prior load's result while bypassing L1 cache accesses. Load reuse enables register file to be used effectively as a larger L1 cache. GPUs have much larger register file than the L1 cache, thus a requested cache line may be evicted from the L1 cache but if the line is read previously the result may be available in the register file. In this case load reuse helps to reduce cache misses, results in both speedup and energy saving due to reduced memory system pressure.

However, to reuse loads, we should prevent memory data hazards that can be caused by stores. In Figure 10, *i1* and *i2* shows read-only memory loads reuse prior results just like arithmetic instructions as stores are not allowed in these memories. However, stores are allowed in scratchpad and global memory. In these memories, when there is a store after a leading load, a trailing load after the store must fetch a newly stored value instead of reusing the old value from the leading load. One solution is to track individual load and store address and dynamically detect hazards, which may be too costly.



#	TBID	WID	Inst (`OP dst, src1, src2`)	Source 1 Values	Source 2 Values	Computation Result Values	Source 1 Reg (Logical → Phy)	Source 2 Reg (Logical → Phy)	Destination Reg (Logical → Phy)
i1	0	0	`ld.const $r1,[$r0],-`	[0, 4, 8, 12]	-	[0, 1, 2, 3]	`$r0→$p6`	–	`$r1→$p1`
i2	1	2	`ld.const $r3,[$r2],-`	[0, 4, 8, 12]	-	***(Reuse i1)***	`$r2→$p6`	–	`$r3→`***$p1***
i3	0	0	`ld.shared $r5,[$r4],-`	[0, 4, 8, 12]	-	[4, 5, 6, 7]	`$r4→$p6`	–	`$r5→$p7`
i4	1	2	`ld.shared $r7,[$r6],-`	[0, 4, 8, 12]	-	[1, 2, 3, 4]	`$r6→$p6`	–	`$r7→$p8`
i5	0	1	`ld.shared $r9,[$r8],-`	[0, 4, 8, 12]	-	***(Reuse i3)***	`$r8→$p6`	–	`$r9→`***$p7***
i6	0	0	`ld.global $r11,[$r10],-`	[0, 4, 8, 12]	-	[10, 20, 30, 40]	`$r10→$p6`	–	`$r11→$p9`
i7	1	2	`ld.global $r10,[$r12],-`	[0, 4, 8, 12]	-	***(Reuse i6)***	`$r12→$p6`	–	`$r13→`***$p9***
i8	0	0	`st.global -,[$r14],$r15`	[0, 4, 8, 12]	[-1, -2, -3, -4]	-	`$r14→$p6`	`$r15→$p10`	–
i9	0	0	`ld.global $r17,[$r16],-`	[0, 4, 8, 12]	-	[-1, -2, -3, -4]	`$r16→$p6`	–	`$r17→$p10`
i10	2	4	`ld.global $r19,[$r18],-`	[0, 4, 8, 12]	-	***(Reuse i6)***	`$r18→$p6`	–	`$r19→`***$p9***

Figure 10: Load instruction reuse examples.

Instead, we conservatively reuse prior loads that are safe from memory data hazards. GPU memory model [32] states that a store is immediately visible only to the thread that executed the store, and the store becomes visible to other threads after the threads are explicitly synchronized using a barrier instruction such as *syncthreads()* or *memfence()*.

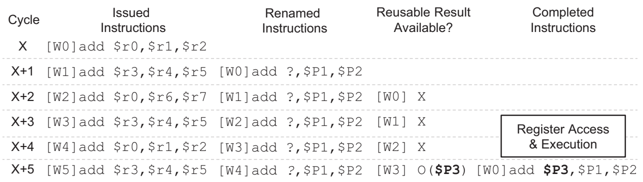
In this model, we can derive two cases where load reuse may cause memory data hazards. First, a warp must not reuse any prior loads as soon as a warp executes a store until reaching the next barrier. This restriction assumes a store immediately causes hazards for all later loads in the warp. Second, a warp must not reuse any loads that executed before the latest barrier. Reusing any result prior to a barrier is potentially unsafe, because other warps may have executed stores but the new values become visible to the other threads after the barrier.

Except the above two cases, loads can reuse previous loads whenever their addresses match. In Figure 10, *i9* assumes *i8* in the same warp 0 may have stored to the same address, thus it does not reuse *i6*. However, *i10* still can reuse *i6* because *i8* is semantically invisible to the different warp 4 until reaching the next barrier.

In addition to the above restrictions, scratchpad loads can reuse only prior loads that are executed by the same thread block because each thread block has a separated scratchpad address space. *i4* cannot reuse *i3* because they read from physically different memory addresses.

We now discuss the changes needed to the reuse buffer design for load reuse. Firstly, reusing scratchpad and global loads is allowed only before a store is executed in the warp. We use two 1-bit store flags for each warp to track whether a scratchpad or a global store is previously executed in that warp. If a renamed instruction is a scratchpad or a global store, then the associated store flag is set. At the reuse stage, if the flag is set, loads bypass reuse table access. The store flag is cleared when the warp reaches a barrier.

Secondly, a load can reuse only prior loads executed after the latest barrier. We track barrier execution count for each thread block. As shown in Figure 9, when updating the reuse



Cycle	Issued Instructions	Renamed Instructions	Reusable Result Available?	Completed Instructions
X	`[W0]add $r0,$r1,$r2`			
X+1	`[W1]add $r3,$r4,$r5`	`[W0]add ?,$P1,$P2`		
X+2	`[W2]add $r0,$r6,$r7`	`[W1]add ?,$P1,$P2`	`[W0] X`	
X+3	`[W3]add $r3,$r4,$r5`	`[W2]add ?,$P1,$P2`	`[W1] X`	
X+4	`[W4]add $r0,$r1,$r2`	`[W3]add ?,$P1,$P2`	`[W2] X`	
X+5	`[W5]add $r3,$r4,$r5`	`[W4]add ?,$P1,$P2`	`[W3] O(`**`$P3`**`)`	`[W0]add `**`$P3`**`,$P1,$P2`

Register Access
& Execution

Figure 11: Execution scenario in the presence of pipeline latency. [W*n*] indicates Warp *n*.

buffer loads record the current count in the *Barrier Count* field. Thus, loads can reuse prior results only from prior loads having the same count. If *Barrier Count* reaches the maximum value, all instructions in the block stops reusing loads until the block is completed. We observed none of our benchmarks executed barrier more than 31 times in a block, thus we used 5 bits for the counter field.

Lastly, scratchpad loads reuse only the previous loads executed by the same thread block. The reuse buffer needs a *Thread Block ID* field, which records the ID of the thread block which executed the instruction. The field is set with a null value for non-scratchpad loads or arithmetic instructions. We used 4 bits for the field, because our baseline GPU allows up to 8 thread blocks per SM and an additional bit is needed to represent a null value.

B. Pending-Retry Mechanism

We observed reusing becomes often inefficient when reusable instructions are issued in a back-to-back fashion. In Figure 11, only the first *add* from needs actual execution as all other instructions can reuse the first result. However, the result of the first *add* is available after 3 cycles later, thus subsequent accesses from X+2 to X+4 miss in the buffer. Although this example uses 2 cycle latency, in practice GPUs have a deeper backend pipeline and execution may take tens of cycles to complete, resulting in even less reuse.

We propose a pending-retry mechanism which allows an instruction to eagerly reserve an entry at a reuse buffer miss. The missed instruction updates all the tag fields and

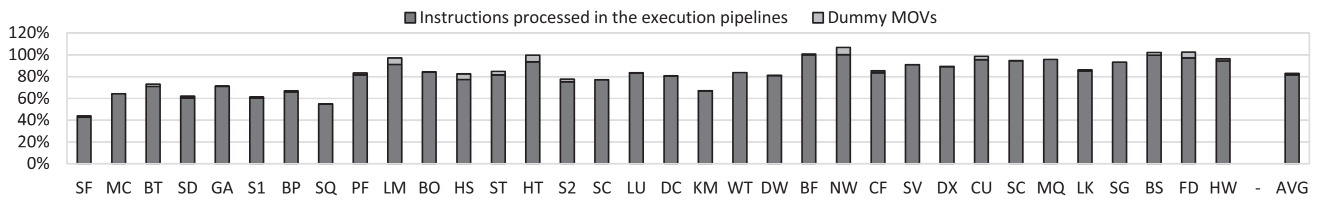


Figure 12: Relative instruction count processed in the backend execution pipeline.

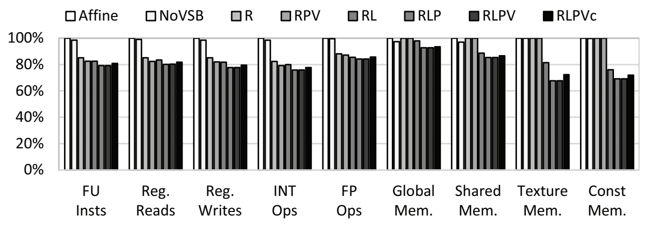


Figure 13: Relative backend operation counts executed by warp instructions.

also sets the pending bit flag in the entry as shown in Figure 9. Thus, subsequent instructions can hit in the buffer and wait in the queue until the pending bit is cleared when the missed instruction updates the result field and retires. Queued instructions can check the pending bit by accessing the reuse buffer when there is no new instruction from the rename stage, and they are re-queued if the result is still pending. We observed using a 16 entry queue helps to generate 15.1% of additional hits, which effect is similar to doubling the reuse buffer entries.

C. Reducing Register Bank Conflicts

Value signature buffer performs verify-read operations instead of register writes for registers that hit in the buffer. We observed that verify-reads can interfere with true read requests and cause performance degradation due to increased bank conflicts. We propose a verify cache which is inspired by register file cache [14]. The verify cache is tagged using physical register IDs and returns the register values. Verify-reads access the cache before they access register banks, and bypass bank access if they hit in the cache. A miss in the cache replaces the cache line after it reads the actual register value from the banks. A register write evicts the associated cache line.

VII. Evaluation

A. Methodology

Benchmarks: Table I lists 34 benchmarks we used from Parboil [34], Rodinia [33], and NVIDIA CUDA SDK [32]. For all benchmarks we used the largest available input sets. Each benchmark execution is sampled up to 1 billion instructions [36] or fully executed.

Table I: List of benchmark applications.

Name	Abbr.	%FP	Suite	Name	Abbr.	%FP	Suite
SobelFilter	SF	6.7%	[32]	MonteCarlo	MC	49.3%	[32]
b+tree	BT	-	[33]	sad	SD	-	[34]
gaussian	GA	2.2%	[33]	srad-v1	S1	15.6%	[33]
backprop	BP	15.0%	[33]	SobolQR	SQ	4.5%	[32]
pathfinder	PF	-	[33]	lbm	LB	54.2%	[34]
binoOpts	BO	30.6%	[32]	hotspot	HS	17.6%	[33]
stencil	ST	9.3%	[34]	hybridsort	HT	17.2%	[33]
srad-v2	S2	25.2%	[33]	scan	SN	-	[32]
lud	LU	19.0%	[33]	dct8x8	DC	34.0%	[32]
kmeans	KM	18.4%	[33]	fastWlshTf	WT	16.1%	[32]
dwt2d	DW	-	[33]	bfs	BF	-	[33]
nw	NW	-	[33]	cfd	CF	62.9%	[33]
spmv	SV	6.3%	[34]	dxtc	DX	43.0%	[32]
cutcp	CU	73.5%	[34]	strmclster	SC	21.9%	[33]
mri-q	MQ	63.9%	[34]	leukocyte	LK	33.4%	[33]
sgemm	SG	68.8%	[34]	BlackSchls	BS	74.4%	[32]
FDTD3d	FD	33.0%	[32]	heartwall	HW	9.2%	[33]

Table II: Simulation parameters.

Parameter	Value
SM parameters	700 MHz, 15 SMs,
	2 schedulers/SM, GTO scheduling
Resource limits/SM	1,024 warp registers (32,768 thread registers)
	48 warps, 8 thread blocks
Register file	128 KB
Scratchpad memory	48 KB
L1 caches	D$: 32 KB, 4-way, 64 MSHR
	T$: 12 KB, C$: 8 KB, I$: 2 KB
NoC	Fully connected, 32 B/direction/cycle @700MHz
L2 cache	6 partitions, 128 KB 8-way @700MHz
	200 cycles latency [35]
DRAM	32 entry scheduling queue, @924MHz
	440 cycles latency [35]
Reuse cache	256 entries (varied)
Value signature buffer	256 entries (varied)
Verify cache	8 entries (varied)

Timing and energy modeling: Our simulator is derived from GPGPU-sim v3.2 [37] and simulates the PTXplus ISA. Energy consumed by additional components are modeled using CACTI [38] and Synopsis Design Compiler with a 45nm library [39], and combined with the GPUWattch [40] results.

Machine models: We use parameters in Table II unless otherwise mentioned. The following is the reuse designs where optimizations are applied incrementally.

- *R*: Minimum reuse design with register renaming, reuse buffer, and value signature buffer.
- *RL*: *R* with load reuse (Section VI-A).
- *RLP*: *RL* with pending-retry (Section VI-B).
- *RLPV*: *RLP* with verify cache (Section VI-C).

We also use the following models for comparison.

- *Base*: The baseline GPU (Section II)

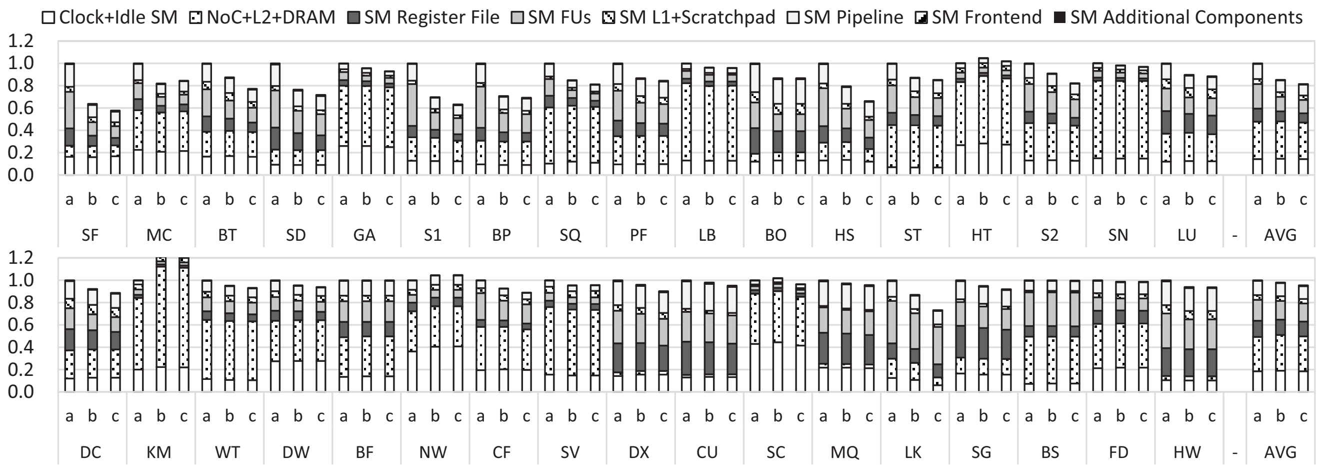


Figure 14: GPU energy consumption breakdown (a:Base, b:RPV, c:RLPV).

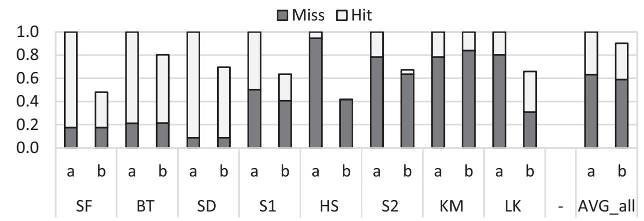


Figure 15: L1 cache access breakdown for selected benchmarks. AVG is the global average for all 48 benchmarks. (a: Base, b:RLPV)

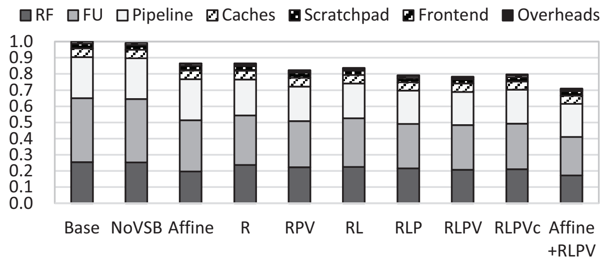


Figure 16: Relative SM energy consumption.

- *RPV*: *RLPV* without load reuse.
- *RLPVc*: *RLPV* with capped-register policy instead of max-register policy (Section V-E).
- *NoVSB*: *R* without the value signature buffer. Instead, a new register is allocated for every convergent register write.
- *Affine*: A hypothetical, energy optimized GPU inspired by previous works [9], [10]. A 1024-bit warp register value is converted to a 64-bit (32-bit base, 32-bit stride) tuple when all adjacent thread register values have the same stride value. An affine tuple is stored to 1 128-bit bank instead of 8 banks so that the tuple consumes only 1/8 bank access energy. Functional units consume energy as much as 1 FU lane if all inputs of an warp instruction are affine tuples and its operation produces an affine tuple with affine inputs, such as *mov, add, sub, mul* operations.
- *Affine+RLPV*: *RLPV* on *Affine* instead of on *Base*.

B. Backend Processing

Figure 12 shows the percentage of backend processed instructions with *RLPV* relative to *Base*. 18.7% of dynamic warp instructions bypassed backend execution. Dummy MOV insertions to handle branch divergence increase instruction count by 1.6% on average.

Figure 13 compares backend operation counts of executed warp instructions. *Affine* executes the same number of warp instructions with *Base* but it saves energy by reducing energy consumed per warp instruction, while our technique saves energy by executing fewer warp instructions. Because *NoVSB* does not correlate register values with register IDs, only less than 2% of instructions was able to bypass backend operation. Compared to *RPV*, *RLPV* reduced up to 32.4% memory pipeline activation with load reuse. Compared to *RLPV*, *RLPVc* shows only slightly less reuse effect, thus we expect substantial energy saving with any one of the register management policy.

C. Energy Consumption

Figure 14 shows the GPU energy breakdown for the *Base*, *RPV*, and *RLPV* models. The height of the bars show the GPU energy consumption relative to *Base*. *RLPV* reduces GPU energy by 10.7% averaged across all benchmarks. As our benchmarks are arranged according to the reusability characterization in Figure 2, the top 24 shows 18.3% which is higher than 4.3% of the bottom 24 benchmarks.

Even without load reuse (*RPV*), 7.6% of energy is saved.

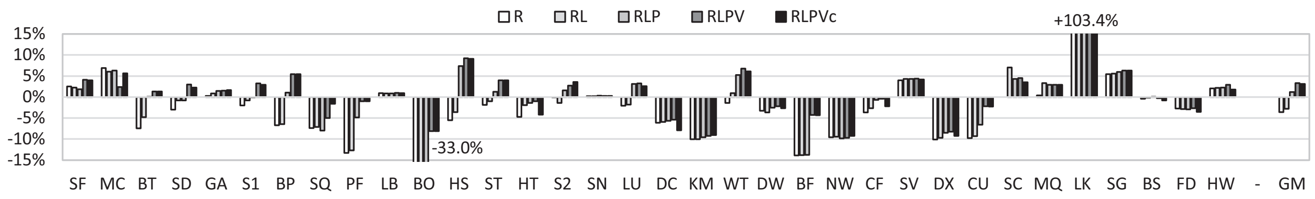
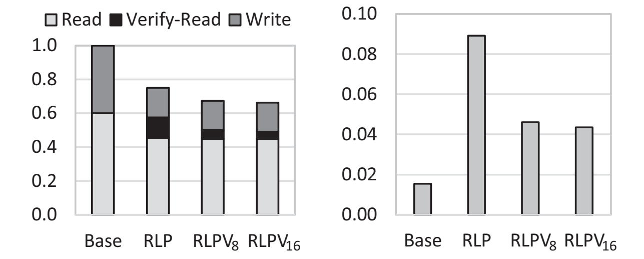


Figure 17: Speedup relative to *Base*.



(a) Relative access breakdown by type.
(b) Bank access retry count per request.

Figure 18: Effects of verify cache on register file. Subscripts indicate the number of cache entries.

However, load reuse enables 3.1% of additional reduction which results in the 10.7% of total reduction. The effect of load reuse was noticeable in benchmarks such as *SF*, *BT*, *HS*, *S2*, and *LK*. Figure 15 shows that both cache accesses and misses are reduced substantially in these benchmarks. Other benchmarks were relatively insensitive, except *KM* where cache misses and the total energy are both increased. As presented in the previous literature [41], [35], [42], *KM* is a highly cache sensitive benchmark; subtle fluctuations in instructions' execution order may have increased cache contention. Warp throttling [41], [35], [43] may alleviate the contention but it is not modeled in our simulation.

Figure 16 compares energy consumed by SM components, which are the primary sources for the reduction. The trend in energy saving is similar to the reductions in backend operations in Figure 13. *RLPV* saves SM energy by 20.5%, which is better than 13.6% of the Affine GPU described in Section VII-A. *Affine+RLPV* saved SM energy by 27.9%, which surpasses all other designs by enabling reuse of non-affine computations.

D. Speedup

Figure 17 shows speedup results compared to the baseline for the 4 reuse designs. We assumed all reuse designs have 4 additional cycle backend delay after the issue stage. Additional performance sensitivity results according to the pipeline delay variations are presented in Section VII-G3. While the majority of applications show small performance variations within 10%, more than 103.2% of speedup was observed in *LK* with *RLPV* which is due to load reuse

Table III: Estimated energy and latency impacts of additional components.

	E/op	Latency	IO Ports	(I,O) bits/op	Max op/inst
Rename table	3.50 pJ	0.33ns	4r 1w	(6, 12)	4r 1w
Reuse buffer table	4.71 pJ	0.31ns	2r 2w	(59, 59)	1r 1w
Hash generation	4.85 pJ	0.95ns	1i 1o	(1024, 32)	1
Val. sig. buf. table	4.96 pJ	0.32ns	2r 2w	(32, 43)	1r 1w
Register allocator	1.35 pJ	0.24ns	1r 1w	(10, 10)	1r 1w
Reference count	0.32 pJ	2.33ns	24i 2o	(10, 10)	6×+1 6×-1
Verify cache	2.93 pJ	0.19ns	2r 2w	(10, 1024)	1r 1w

worked extremely well and lead to L1 cache misses by 61.5%, as shown in Figure 15.

Without verify cache (*RLP*), applications such as *GA*, *BO*, and *BF* suffer from severe performance degradation, which was mitigated with verify cache as shown in the *RLPV* results. Figure 18a shows 48% of writes are substituted to verify-reads in *RLP*. However, as shown in Figure 18b bank conflicts are more frequent in *RLP* than in the baseline. We observe a small 8-entry verify cache $RLPV_8$ removes nearly 50% of the increased bank conflicts. Even the size is doubled the improvement was marginal.

E. Hardware Costs

Table III summarizes estimated energy and latency impacts of the additional components. Our baseline GPU can issue two warp instructions at a cycle independently from two groups of 24 warps each. To rename one warp instruction issued from a group for every cycle, 48 rename tables are split into two 24×63-entry tables where each table has 4r1w ports. Both the reuse buffer and the value signature have 2r2w ports because up to two warp instructions can access the tables at a cycle. The register allocator is modeled as a queue using a 1r1w table. The reference counting system has $12 \times +1$ and 12×-1 physical register ID inputs.

Based on the latency estimation, we assumed new stages add 4 cycles to each warp instruction's processing latency but does not decrease the baseline clock frequency, which is 0.7 GHz or 1.43 ns per cycle. The rename and reuse stages take 1 cycle each, and the register allocation stage is further pipelined into 2 cycles for hash generation and table access. For hash generation, we used the H3 hash function [44], [45]. In our implementation, a hash value is generated using hardwired and cascaded XOR gates associated with each hash output bit, and there are 13 XOR gates in the critical path. Assuming each gate takes 7 ps, we estimated a hash

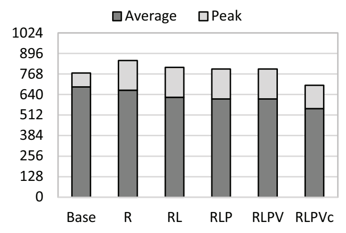


Figure 19: Physical register utilization.

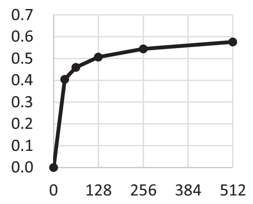


Figure 20: Value sig. buf. entry vs. hit rate.

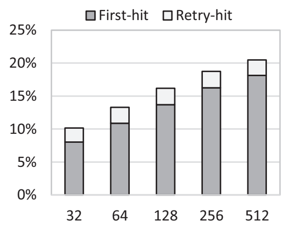


Figure 21: Reuse buffer entry vs. % insts bypass FU execution.

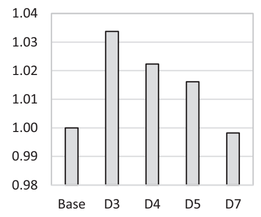


Figure 22: Backend pipeline delay vs. speedup.

value can be calculated in 1 cycle. Reference counting takes 2 cycles, but it can be performed in parallel to instruction processing, thus we did not include this in the increased instructions' latency.

We estimate the total storage required per SM is about 9.9 KB, which is 4.3% of the 230 KB storage for the register file, the L1 caches, and the scratchpad memory combined. Each of the 48 rename tables has 63 entries with 12 bit per entry, thus the total size is 4.42 KB. The table in the reuse buffer has 256 entries of 59 bits, thus the total size is 1.84 KB. A value signature buffer entry is 43-bit wide, thus 256 entries use 1.34 KB. The verify cache has 1035 bit per entry, thus 8 entries occupy 1.01 KB. The reference counting system has 1024 10-bit counters for each physical register, thus total size is 1.25 KB. Other structures need trivial storage costs.

F. Impact on Register Utilization

Figure 19 compares the utilization of physical registers averaged on all benchmarks. Each bar shows how many of 1024 physical registers are used. The *average* part shows the average utilization and the *peak* shows the maximum utilization during the benchmark run. Even *Base* does not fully utilize registers because thread parallelism is often capped by other limiting factors such as thread count or scratchpad memory capacity [31], [46]. Note that *RLPVc* uses the capped-register policy which restricts the number of physical registers in use within the number of logical registers for all active warps. However, even *RLPV*, which does not have this restriction, shows less average utilization than *Base*, because our register reuse enables multiple logical registers to share a physical register, thus fewer physical registers are needed compared to the one-to-one register mapping.

G. Design Space Exploration

1) Sensitivity to Value Signature Buffer Size: Figure 20 shows the hit rate of the value signature buffer averaged across all benchmarks when the number of entries is varied from 0 to 256. Even with relatively a smaller value buffer size of 128, more than 50% of hits were generated. Because the hit rate starts to saturate beyond 256 entries, we used 256-entries for our default configuration.

2) Sensitivity to Reuse Buffer Size: Figure 21 shows the percentage of instructions reused previous results when the reuse buffer's entry count is varied from 32 to 512. As the entry count increases, the buffer can capture prior results better. We chose 256 entries as our default configuration which showed 18.7% hit rate. With 512 entries more than 20% instructions hit in the buffer. The bars also show the fraction from pending-retry hits discussed in Section VI-B. Pending-retry increased hit rate as much as doubling the entry count, thus the effect is clearly significant.

3) Impacts of Increased Backend Latency: In Section VII-E, we estimated the additional stages for reuse introduce 4 additional cycles. Figure 22 shows the relative speedup when the latency is varied from 3 cycles to 7 cycles. *Dn* in the X-axis denotes *n* cycle delay, thus *D4* shows the result with our 4 cycle default configuration. Beyond 7 cycle the performance becomes lower than the baseline. However, even with this worst case assumption reuse did not severely degrade the performance.

VIII. Related Work

This work is inspired by instruction reuse designs studied for CPUs [22], [23], [47], [26], [27], [24], [29]. However, to our knowledge this work is the first paper that explored instruction reuse adoption to GPUs.

One motivation of instruction reuse is to salvage discarded results when recovering from branch mispredictions. Squash reuse allows instructions to reuse only its own result that generated in the mispredicted program flow [22], but it is not the case in GPUs due to the lack of speculation. General instruction reuse relaxes the above restriction by allowing instructions to reuse results produced by other instructions [26]. Our reuse design is based on the concept of general reuse.

Several reuse designs have been proposed for CPUs [23], [27], [48], [29]. Our reuse buffer stores physical register IDs instead of values for energy efficient operation, thus its design is similar to the integration table [47], [26], [24], but conceptually more similar to the *Sv* design in [22]. Our goal is how to design this type of reuse efficiently for GPUs.

Warp register reuse is more efficient than a previous register reuse design [27]. In conventional CPU renaming, a physical register ID is used both as a tag for instruction scheduling and as an address for the register file. For correct dependency resolution, tags (physical registers) must be allocated at rename where instructions' program order can be tracked. In CPUs, if a physical register is reassigned at writeback, the new tag must be re-broadcast to dependent instructions in the issue queue, which introduces significant complexity in design [27]. However, in our GPU case, instruction scheduling remains irrelevant to renaming as the scheduling is performed using logical registers as in the baseline GPU. Thus, without such re-broadcasts we can remap a logical register to a physical register. We also designed a value signature buffer which leverages hashing to achieve reuse within feasible energy and storage cost.

The remapping of logical registers is inspired by recent GPU studies [31], [49]. Register virtualization [31] shrinks register usage by re-allocating a physical register to another warp as soon as the last read to the register completes, instead of waiting for the register to be released until the thread block's completion. Warp register reuse also reduces register utilization by enabling multiple logical registers can share a physical register. Virtual threads [46] and Zorua [50] generalize this remapping concept further to other GPU resources to improve resource utilization. These techniques enable various optimizations such as to alleviating performance cliffs or improving performance portability. However, our focus is more on improving efficiency of computation itself by reducing energy per processed instruction.

In CPUs, there are even more aggressive renaming-based optimizations combined with speculation, such as value prediction [51], [52], [53], [54], [55], [56]. Instead of waiting for a computation to complete, the predicted result is used to continue execution in a speculative state. Memory bypassing [23], [47], [57] speculatively replace memory dependence into register dependence. Because GPUs lack of speculation, adding verification and recovery logic only for this purpose may introduce substantial overhead. Instead, we proposed to conservatively reuse previous loads that are free from memory data hazards in the GPU memory model.

Warp instruction reuse can be combined with existing techniques that improve energy efficiency or performance, such as more efficient register files [14], [15], [16], [17], [21], pipeline design [18], [19], warp scheduling [58], [41], [59], [35], [60], or memory system optimizations [61], [62], [42], [63]. Recent GPU studies have further optimized backend energy by using more efficient data encoding. For instance, scalarization [5], [64], [6], affine vector conversion [9], [10], and compression [20] use fewer bits to represent a 1024-bit warp register value. Energy can be saved if computation can be performed in the compressed form without reconversion to the original 1024-bit representation, as explored in scalar [5], [6], [7], [8], [13] and affine [9], [10], [12], and compressed [11] execution. These works have focused on spatial redundancy for reducing energy per executed warp instruction, while our reuse technique more focuses on temporal redundancy in repeated computations for executing fewer warp instructions. The *Affine+RLPV* model in the evaluation shows an example of synergistic energy reduction effect when both types of redundancies are considered at the same time.

There have been memoization-based approaches at various levels. For example, memoization techniques specific for graphics workloads [65] has been explored, while our technique focus more on general workloads. Such memoization can be performed at the API level [66] that is much coarser than our design. The proposed design performs reuse at the instruction level.

IX. Conclusion

We proposed to improve GPU energy efficiency by allowing warp instructions to reuse previous computation results. The proposed warp instruction reuse enables to bypass energy hungry backend operations such as register accesses and functional unit activations. We also proposed warp register reuse that enhances warp instruction reuse by eliminating additional value operations required for reuse. We also investigated various aspects including performance and energy impacts using a wide range of design choices.

Acknowledgment

This work is supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP) (No. NRF-2015R1A2A2A01008281) and by the ICT R&D program of MSIP/IITP. [No. 2016-0-00140, Development of Application Program Optimization Tools for High Performance Computing Systems]

References

[1] NVIDIA. Whitepaper: NVIDIA Fermi.
[2] NVIDIA. Whitepaper: NVIDIA Kepler GK110.
[3] NVIDIA. NVIDIA GeForce GTX 980 Whitepaper.
[4] NVIDIA. GP100 Pascal Whitepaper.
[5] M. Mantor and M. Houston, "Amd graphics core next," *AMD Fusion Developer Summit*, 2011.
[6] Y. Yang, P. Xiang, M. Mantor, N. Rubin, L. Hsu, Q. Dong, and H. Zhou, "A case for a flexible scalar unit in simt architecture," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, May 2014, pp. 93–102.
[7] Z. Chen, D. Kaeli, and N. Rubin, "Characterizing scalar opportunities in gpgpu applications," in *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*. IEEE, 2013, pp. 225–234.
[8] Z. Chen and D. Kaeli, "Balancing scalar and vector execution on gpu architectures," in *Parallel and Distributed Processing Symposium, 2016 IEEE International*. IEEE, 2016, pp. 973–982.

[9] S. Collange, D. Defour, and Y. Zhang, "Dynamic detection of uniform and affine vectors in gpgpu computations," in *European Conference on Parallel Processing*. Springer, 2009, pp. 46–55.

[10] J. Kim, C. Torng, S. Srinath, D. Lockhart, and C. Batten, "Microarchitectural mechanisms to exploit value structure in simt architectures," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. ACM, 2013, pp. 130–141.

[11] S. Lee, K. Kim, G. Koo, H. Jeon, M. Annavaram, and W. W. Ro, "Improving energy efficiency of gpus through data compression and compressed execution," *IEEE Transactions on Computers*, vol. PP, no. 99, pp. 1–1, 2016.

[12] K. Wang and C. Lin, "Decoupled affine computation for simt gpus," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. ACM, 2017, pp. 295–306.

[13] 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)*, ser. HPCA '17, Feb 2017, pp. 601–612.

[14] 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 *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ser. ISCA '11. ACM, 2011, pp. 235–246.

[15] M. Gebhart, S. W. Keckler, B. Khailany, R. Krashinsky, and W. J. Dally, "Unifying primary cache, scratch, and register file memories in a throughput processor," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-45. IEEE Computer Society, 2012, pp. 96–106.

[16] M. Abdel-Majeed and M. Annavaram, "Warped register file: A power efficient register file for gpgpus," in *Proceedings of the 2013 IEEE 19th International Symposium on High-Performance Computer Architecture*, ser. HPCA '13, 2013.

[17] M. Abdel-Majeed, D. Wong, and M. Annavaram, "Warped gates: Gating aware scheduling and power gating for gpgpus," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-46. ACM, 2013, pp. 111–122.

[18] S. Z. Gilani, N. S. Kim, and M. J. Schulte, "Exploiting gpu peak-power and performance tradeoffs through reduced effective pipeline latency," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-46. ACM, 2013, pp. 74–85.

[19] S. Z. Gilani, N. S. Kim, and M. J. Schulte, "Power-efficient computing for compute-intensive gpgpu applications," in *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*. IEEE, 2013, pp. 330–341.

[20] S. Lee, K. Kim, G. Koo, H. Jeon, W. W. Ro, and M. Annavaram, "Warped-compression: Enabling power efficient gpus through register compression," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ser. ISCA '15. ACM, 2015, pp. 502–514.

[21] 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)*, ser. HPCA '17, Feb 2017, pp. 589–600.

[22] A. Sodani and G. S. Sohi, "Dynamic instruction reuse," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ser. ISCA '97. ACM, 1997, pp. 194–205.

[23] S. Jourdan, R. Ronen, M. Bekerman, B. Shomar, and A. Yoaz, "A novel renaming scheme to exploit value temporal locality through physical register reuse and unification," in *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, ser. MICRO 31. IEEE Computer Society Press, 1998, pp. 216–225.

[24] V. Petric, T. Sha, and A. Roth, "Reno: A rename-based instruction optimizer," in *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, ser. ISCA '05. IEEE Computer Society, 2005, pp. 98–109.

[25] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco, "Gpus and the future of parallel computing," *IEEE Micro*, vol. 31, no. 5, pp. 7–17, 2011.

[26] V. Petric, A. Bracy, and A. Roth, "Three extensions to register integration," in *Microarchitecture, 2002.(MICRO-35). Proceedings. 35th Annual IEEE/ACM International Symposium on*. IEEE, 2002, pp. 37–47.

[27] S. Balakrishnan and G. S. Sohi, "Exploiting value locality in physical register files," in *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*. IEEE, 2003, pp. 265–276.

[28] A. Roth, "Physical register reference counting," *IEEE Computer Architecture Letters*, vol. 7, no. 1, pp. 9–12, Jan 2008.

[29] A. Perais *et al.*, "Cost effective physical register sharing," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2016, pp. 694–706.

[30] S. Battle, A. D. Hilton, M. Hempstead, and A. Roth, "Flexible register management using reference counting," in *IEEE International Symposium on High-Performance Comp Architecture*, Feb 2012, pp. 1–12.

[31] H. Jeon, G. S. Ravi, N. S. Kim, and M. Annavaram, "Gpu register file virtualization," in *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-48, 2015.

[32] NVIDIA. CUDA C Programming Guide.

[33] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization*, ser. IISWC '09, Oct 2009, pp. 44–54.

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

[35] A. Sethia, D. Jamshidi, and S. Mahlke, "Mascar: Speeding up gpu warps by reducing memory pitstops," in *Proceedings of the 2015 IEEE 21st International Symposium on High Performance Computer Architecture*, ser. HPCA '15, 2015.

[36] A. Jog, O. Kayiran, N. Chidambaram Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "Owl: Cooperative thread array aware scheduling techniques for improving gpgpu performance," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13. ACM, 2013, pp. 395–406.

[37] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, ser. ISPASS '09, April 2009, pp. 163–174.

[38] D. Tarjan, S. Thoziyoor, and N. P. Jouppi, "Cacti 4.0,"

Technical Report HPL-2006-86, HP Laboratories Palo Alto, Tech. Rep., 2006.

[39] J. Knudsen, “Nangate 45nm open cell library.”

[40] 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 *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA ’13. ACM, 2013, pp. 487–498.

[41] T. G. Rogers, M. O’Connor, and T. M. Aamodt, “Cache-conscious wavefront scheduling,” in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-45. IEEE Computer Society, 2012, pp. 72–83.

[42] Y. Oh, K. Kim, M. K. Yoon, J. H. Park, Y. Park, W. W. Ro, and M. Annavaram, “Apres: Improving cache efficiency by exploiting load characteristics on gpus,” in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, June 2016, pp. 191–203.

[43] O. Kayiran, A. Jog, M. T. Kandemir, and C. R. Das, “Neither more nor less: Optimizing thread-level parallelism for gpgpus,” in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT ’13. IEEE Press, 2013, pp. 157–166.

[44] M. V. Ramakrishna, E. Fu, and E. Bahcekapili, “Efficient hardware hashing functions for high performance computers,” *IEEE Transactions on Computers*, vol. 46, no. 12, pp. 1378–1381, Dec 1997.

[45] D. Sanchez, L. Yen, M. D. Hill, and K. Sankaralingam, “Implementing signatures for transactional memory,” in *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, Dec 2007, pp. 123–133.

[46] M. K. Yoon, K. Kim, S. Lee, W. W. Ro, and M. Annavaram, “Virtual thread: Maximizing thread-level parallelism beyond gpu scheduling limit,” in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, June 2016, pp. 609–621.

[47] A. Roth and G. S. Sohi, “Register integration: a simple and efficient implementation of squash reuse,” in *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*. ACM, 2000, pp. 223–234.

[48] L. Tran, N. Nelson, F. Ngai, S. Dropsho, and M. Huang, “Dynamically reducing pressure on the physical register file through simple register sharing,” in *Performance Analysis of Systems and Software, 2004 IEEE International Symposium on-ISPASS*. IEEE, 2004, pp. 78–87.

[49] K. Kim, S. Lee, M. K. Yoon, G. Koo, W. W. Ro, and M. Annavaram, “Warped-preexecution: A gpu pre-execution approach for improving latency hiding,” in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, March 2016, pp. 163–175.

[50] N. Vijaykumar, K. Hsieh, G. Pekhimenko, S. Khan, A. Shrestha, S. Ghose, A. Jog, P. B. Gibbons, and O. Mutlu, “Zorua: A holistic approach to resource virtualization in gpus,” in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2016, pp. 1–14.

[51] M. H. Lipasti and J. P. Shen, “Exceeding the dataflow limit via value prediction,” in *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, ser. MICRO 29. IEEE Computer Society, 1996, pp. 226–237.

[52] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen, “Value locality and load value prediction,” in *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS VII. ACM, 1996, pp. 138–147.

[53] A. Sodani and G. S. Sohi, “Understanding the differences between value prediction and instruction reuse,” in *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, ser. MICRO 31. IEEE Computer Society Press, 1998, pp. 205–215.

[54] P. Marcuello, J. Tubella, and A. González, “Value prediction for speculative multithreaded architectures,” in *Proceedings of the 32Nd Annual ACM/IEEE International Symposium on Microarchitecture*, ser. MICRO 32. IEEE Computer Society, 1999, pp. 230–236.

[55] S. Liu, C. Eisenbeis, and J.-L. Gaudiot, “Value prediction and speculative execution on gpu,” *International Journal of Parallel Programming*, vol. 39, no. 5, pp. 533–552, 2011.

[56] A. Perais and A. Seznec, “Eole: Paving the way for an effective implementation of value prediction,” in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA ’14. IEEE Press, 2014, pp. 481–492.

[57] G. H. Loh, R. Sami, and D. H. Friendly, “Memory bypassing: Not worth the effort,” in *Proc. 1st Workshop on Duplicating, Deconstructing, and Debunking*, 2002, pp. 71–80.

[58] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt, “Improving gpu performance via large warps and two-level warp scheduling,” in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44. ACM, 2011, pp. 308–317.

[59] Q. Xu and M. Annavaram, “Pats: Pattern aware scheduling and power gating for gpgpus,” in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, ser. PACT ’14. ACM, 2014, pp. 225–236.

[60] S.-Y. Lee, A. Arunkumar, and C.-J. Wu, “Cawa: Coordinated warp scheduling and cache prioritization for critical warp acceleration of gpgpu workloads,” in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ser. ISCA ’15. ACM, 2015, pp. 515–527.

[61] J. Lee, N. B. Lakshminarayana, H. Kim, and R. Vuduc, “Many-thread aware prefetching mechanisms for gpgpu applications,” in *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-43. IEEE Computer Society, 2010, pp. 213–224.

[62] R. Ausavarungnirun, S. Ghose, O. Kayiran, G. H. Loh, C. R. Das, M. T. Kandemir, and O. Mutlu, “Exploiting inter-warp heterogeneity to improve gpgpu performance,” in *2015 International Conference on Parallel Architecture and Compilation (PACT)*, ser. PACT ’15,, Oct 2015, pp. 25–38.

[63] G. Koo, Y. Oh, W. W. Ro, and M. Annavaram, “Access pattern-aware cache management for improving data utilization in gpu,” in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA ’17. ACM, 2017, pp. 307–319.

[64] K. Asanovic, S. W. Keckler, Y. Lee, R. Krashinsky, and V. Grover, “Convergence and scalarization for data-parallel architectures,” in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE Computer Society, 2013, pp. 1–11.

[65] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis, “Eliminating redundant fragment shader executions on a mobile gpu via hardware memoization,” in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA ’14. IEEE Press, 2014, pp. 529–540.

[66] H. Zhou, Y. Fu, and C. Liu, “Supporting dynamic GPU computing result reuse in the cloud,” in *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*. USENIX Association, 2015.