RegLess: Just-in-Time Operand Staging for GPUs

John Kloosterman
University of Michigan
jklooste@umich.edu

Jonathan Beaumont
University of Michigan
jbbeau@umich.edu

D. Anoushe Jamshidi*
University of Michigan
ajamshid@umich.edu

Jonathan Bailey
University of Michigan
jbaile@umich.edu

Trevor Mudge
University of Michigan
tnm@umich.edu

Scott Mahlke
University of Michigan
mahlke@umich.edu

ABSTRACT

The register file is one of the largest and most power-hungry structures in a Graphics Processing Unit (GPU), because massive multithreading requires all the register state for every active thread to be available. Previous approaches to making register accesses more efficient have optimized how registers are stored, but they must keep all values for active threads in a large, high-bandwidth structure. If operand storage is to be reduced further, there will not be enough capacity for every live value to be stored at the same time. Our insight is that computation graphs can be sliced into regions and operand storage can be allocated to these regions as they are encountered at run time, allowing a small operand staging unit to replace the register file. Most operand values have a short lifetime that is contained in one region, so their value does not need to persist in the staging unit past the end of that region. The small number of longer-lived operands can be stored in lower-bandwidth global memory, but the hardware must anticipate their use to fetch them early enough to avoid stalls. In RegLess, hardware uses compiler annotations to anticipate warps' operand usage at run time, allowing the register file to be replaced with an operand staging unit 25% of the size, saving 75% of register file energy and 11% of total GPU energy with no average performance loss.

CCS CONCEPTS

• **Computer systems organization** → *Single instruction, multiple data*; • **Software and its engineering** → *Compilers*;

KEYWORDS

GPU, register file, GPU compiler

ACM Reference format:
John Kloosterman, Jonathan Beaumont, D. Anoushe Jamshidi, Jonathan Bailey, Trevor Mudge, and Scott Mahlke. 2017. RegLess: Just-in-Time Operand Staging for GPUs. In *Proceedings of MICRO-50, Cambridge, MA, USA, October 14–18, 2017,* 14 pages.
https://doi.org/10.1145/3123939.3123974

*Now at Apple, Inc.

MICRO-50, October 14–18, 2017, Cambridge, MA, USA

ACM ISBN 978-1-4503-4952-9/17/10…$15.00
https://doi.org/10.1145/3123939.3123974

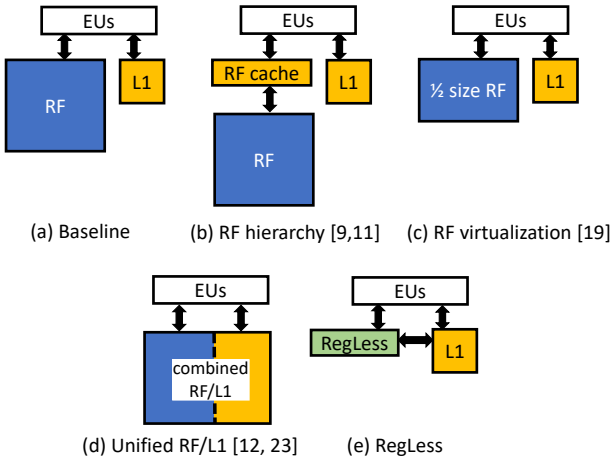

Figure 1: Comparison of GPU register energy reduction techniques that change how execution units (EUs) read operands from the register file (RF)

1 INTRODUCTION

As Graphics Processing Units (GPUs) proliferate from gaming desktops into datacenter and mobile environments, they are required to be more energy-efficient than ever before. GPUs' high computational throughput comes from their massively multithreaded architecture, where stalls in one thread are hidden by switching to another thread and many threads can issue each cycle. This requires the GPU to store the context for every active thread in a way that makes it available at any time.

Since registers make up most of each thread's state, GPUs have very large register files. To store the registers for the 32 SIMD lanes for each of the 64 hardware threads (called warps), each core (called an SM) in NVIDIA's Maxwell architecture has a 256KB register file. Because of its size, on GPU architectures similar to the GTX 980, the register file consumes up to 13% of total GPU power, nearly as much as the arithmetic units or DRAM [33]. As GPU designs provision more concurrency, the register file will only grow. Therefore, reducing the size of the register file and the energy used to access it is an important part of making GPUs more efficient.

Previous approaches have focused on optimizing register storage space, as shown in Figure 1. The baseline (a) reads all operands from the large register file (RF) and has a separate L1 data cache. By adding a smaller hardware [9] or software [11] managed cache in front of the main register file (b), most register accesses can

be filtered by a smaller structure. By dynamically reusing register capacity otherwise used to store dead values, a smaller register file (c) can be provisioned or portions of the register file can be power gated [19]. For applications that do not use the entire register file, portions of the register file can be used as more L1 cache or scratchpad memory (d) to increase occupancy and performance [12, 23].

Our technique, RegLess (e), reduces the amount of storage space by anticipating when registers will be used in time. Instead of a full register file that contains every live value, RegLess maintains a small operand staging unit. Code running on the GPU is divided into regions and just in time for a region to begin execution RegLess allocates space for it in the staging unit. Most operands' lifetimes are contained in one region, so when that region is finished executing, the staging unit can reuse their storage. An operand value with a lifetime that spans regions can be evicted into the memory hierarchy when no active region is using it, so before a region can begin executing, RegLess fetches any needed long-lived registers from memory.

In order for the hardware to manage the operand staging unit effectively, it needs visibility into future register usage, which is provided by the compiler with annotations in the instruction stream. A hardware resource manager uses these annotations to anticipate which registers a warp is about to access. The resource manager also controls which warps are eligible to issue instructions, ensuring the warps allowed to execute always have their registers ready in the staging unit. Other annotations inform the hardware when a register dies and can be erased from the staging unit or memory system.

Only a few registers can be transferred between the staging unit and memory without incurring performance loss. Because the L1 data cache in each SM can only service one request per cycle, the bandwidth available to fill the staging unit is much smaller than the bandwidth needed to service register reads and writes. To address this, the RegLess compiler divides regions at the points that maximize the number of registers interior to one region, as the values in these registers will never be transferred to or from memory. By creating regions that rarely need their operands fetched from memory and managing staging unit capacity for these regions in hardware, RegLess can maintain performance while vastly reducing register storage.

Our contributions in this work include:

- Replacing the GPU register file with a small operand staging unit that only holds values about to be accessed.
- Designing compiler techniques for dividing kernels into regions that maximize the number of registers interior to a region.
- Detailing hardware components for managing operand storage capacity, fetching operands from memory just before they will be used, and minimizing the performance impact of storing register values in memory.
- Analyzing the power and area required by RegLess with a placed-and-routed Verilog model.
- Demonstrating that the RegLess system can reduce register capacity by 75% with no average performance loss.

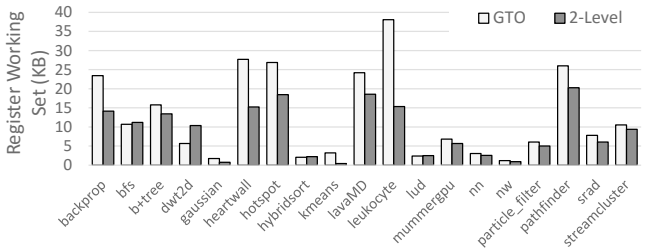

Figure 2: Average register working set in 100 cycle window for GTO and 2-level warp schedulers in baseline 2048 KB register file for benchmarks in Rodinia [6]

2 RF REPLACEMENT CHALLENGES

Replacing the register file with an operand staging unit smaller than needed to hold all live registers presents several difficult challenges. Managing the staging unit must be done precisely, as exactly the right operands need to be present in the staging unit at exactly the right time or performance will suffer as warps stall for operands to become available. A first challenge is determining how much of the staging unit each warp will have access to. Another comes from the limited memory bandwidth available for moving values in and out of the staging unit. A final challenge is conserving memory system capacity to allow more cross-region registers to fit in the L1 cache.

2.1 Capacity Allocation

Because of the small capacity of the staging unit (25% of the baseline register file or less), only a subset of registers can be stored in it at any one time. The staging unit will hold fewer registers than might be live across all active warps, so not every warp can have all its live registers present in it. However, because not all registers are accessed by every warp all the time, there is an opportunity to store only the subset of registers that will be used in an interval of time. Figure 2 shows the register capacity accessed in a 100-cycle window in each Rodinia [6] benchmark. For most applications, this is 10% or less of the baseline register file's 2048 KB capacity.

One approach to allocating capacity would be with standard spills and fills inserted by the compiler. Each warp would have an allocation in the staging unit that it would manage using load and store instructions. This strategy fails to take into account that warps are not equally likely to issue instructions – dynamically, some warps will be stalled for long-latency operations and their space in the staging unit would be better used by active warps. Another approach would be modelling the staging unit after a cache, allocating space based on which registers are most recently accessed. Although this works when the backing store for the cache is the main register file, this reactive strategy would cause stalls for register fetches if the cache was backed by main memory.

To allocate staging unit capacity only to active warps, RegLess coordinates the warps eligible to issue instructions with the warps that are allocated space in the staging unit. Figure 2 shows that the two-level warp scheduler from [9] reduces the amount of register space that is used in each interval relative to the baseline GTO by scheduling instructions from only a subset of warps at a time. Extending this insight, RegLess only allows warps that have an allocation in the staging unit to issue instructions. In this way, all allocated space is useful to a running warp.

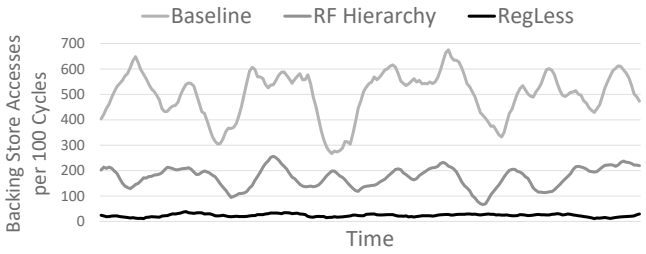

Figure 3: Accesses to the register backing store per 100 cycles during the steady state of hotspot for baseline, RF hierarchy [11] with 8-entry scratchpad, and RegLess with 8 entries per warp

In order to know how much capacity to allocate to each running warp, the RegLess hardware receives information from the compiler. Hardware by itself cannot know how much capacity each warp will use, but the compiler has a global perspective of exactly which registers will be accessed at which points in the program. The best allocation decision is a combination of the hardware's dynamic perspective of how much staging unit capacity is available and the compiler's global perspective of warps' future needs. The RegLess compiler divides a kernel into atomic regions, and the beginning of the region is annotated with how much capacity that region requires in the staging unit in order to run. A hardware resource scheduler activates warps when their next region is allocated capacity in the staging unit. In this way, RegLess anticipates warps' future resource needs in its allocation decision.

2.2 Memory Side Bandwidth

The second problem with eliminating the register file is that the backing store, global memory through the L1, has limited bandwidth. In our model, only one access can be made to the global memory system through the L1 cache per cycle. This is also more constrained than previous work, which has recourse to a main register file with full bandwidth [9, 11]. In order for this not to limit performance, fewer than one request per cycle can be made to transfer a register in or out of the staging unit.

To reduce the number of accesses made to L1, the RegLess compiler creates regions with as many interior registers as possible. *Input* and *output* registers hold values used to communicate between regions, whereas the lifetime of an *interior* register lies entirely inside one region. By guaranteeing each region the space it needs in the staging unit while it executes, any interior registers whose lifetime is contained in the region will never need to be transferred in or out of it. The only registers that must be transferred in or out of the staging unit to the L1 are inputs and outputs – the values communicated between regions. Therefore, when the compiler decides where to put the boundaries between regions, it chooses points with the fewest number of live registers.

The other part of the solution is loading each regions' input registers into the staging unit sufficiently early that instructions do not stall waiting for their registers. Instead of loading registers from L1 when they are first accessed, all the input registers for a region are fetched before any instructions from that region can be issued. We call this register fetching process *preloading*. The staging area needs enough capacity that several warps can be issuing from their regions while other warps preload registers for their next region. Output registers can stay in the staging unit until evicted, so RegLess prefers activating a region from a recently active warp in case an input to the new region was an output of a recent one.

Together, these strategies mean there are far fewer requests made to the backing store than in previous work. Figure 3 compares the number of accesses made to the main register file in the baseline to the accesses made to the large register file in [11] and the accesses made to the L1 cache in RegLess with the same capacity. Because so few accesses filter through RegLess to the L1, on average 0.9%, it becomes feasible to use the low-bandwidth L1 to store cold registers.

2.3 L1 Cache Capacity

A final problem is that the staging unit and the L1 combined are smaller than the register working set for many kernels. Because of this, registers and data in L1 would contend for space in L1 and registers may be evicted across the interconnect to L2 or DRAM. It would take hundreds of cycles to fetch these registers and they would contend for scarce L2 bandwidth. Previous work [31, 41] has recognized that many registers hold values that have similar values for each 4-byte contribution from each lane. This makes registers amenable to compression. RegLess compresses registers that are evicted from the staging unit to the memory system, matching them against fixed patterns that are intentionally simpler than full register file compression techniques, in order to fit more register values in the limited L1 capacity.

3 DESIGN OVERVIEW

In these ways, RegLess' design overcomes the challenges of eliminating the register file using hardware capacity management guided by compiler annotations. To further demonstrate how RegLess operates, we will walk through each component of the system.

First, at compile time (part ① in Figure 4), the kernel is divided into regions of instructions. The compiler annotates the input and output registers of each region. Since the vast majority of registers are intermediates with short lifetimes, these regions have a small number of input and output registers compared to the number of registers which are both produced and consumed inside the region.

At run time, registers are stored in a *operand staging unit* (OSU), with space allocated based on compiler annotations on the regions. When a warp starts running a new region, that region's input registers need to be assembled in the OSU ②. The OSU may already contain some of these registers, and the others will be loaded from the L1 data cache. Not all of a warp's registers are loaded – only the ones that will be used in the next region. The instructions in a region are guaranteed to have the registers they need available in the OSU as they execute. As values are used for the last time in the region, they are erased from the cache or marked for eviction ③.

RegLess orchestrates this process by actively managing the OSU capacity. A *capacity manager* (CM) ④ makes warps eligible to issue instructions only when all the warp's input registers are present and there is space for all the warp's interior registers in the OSU. As warps complete regions, their registers are reclaimed and the CM uses the free capacity to preload registers for a new region. The register working set often fits in the OSU, and any overflow almost always fits in the L1 data cache and does not generate traffic at lower levels of the memory hierarchy.

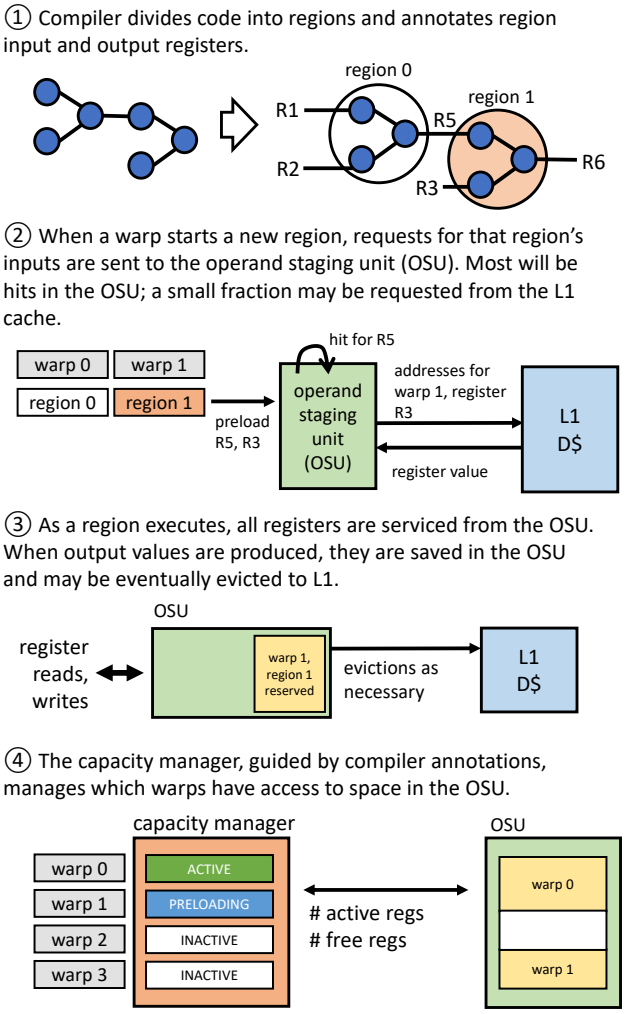

Figure 4: Walkthrough

Next, we will describe the compiler techniques and hardware implementation of RegLess.

4 COMPILER CODE GENERATION

In order for the hardware to make register allocation decisions, it needs to know which registers to move into the staging unit and when those registers will no longer be needed. The compiler provides this through metadata inserted in the instruction stream, as it has whole-program visibility into when each register will be used. In order to do this, the compiler divides the kernel into regions of instructions and annotates each region with data about which registers must be present to start the region, the number of temporaries used in the region, and when the regions' registers can be erased or evicted from the staging unit and memory system.

4.1 Region Creation

Where the compiler chooses to create region boundaries affects how much data movement is necessary when running a kernel. Registers that are produced outside the region but used inside it need to be fetched from memory before the region can start running, but registers with their lifetime entirely within one region are guaranteed to never be transferred to memory, as regions are scheduled atomically by the RegLess hardware. Therefore, the compiler should draw region boundaries to minimize the number of registers communicated between regions and maximizing registers interior to a region.

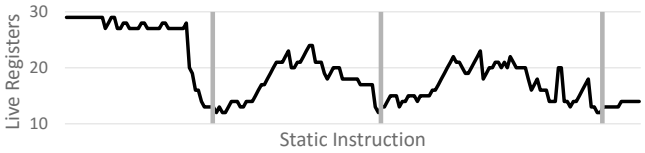

Figure 5: Count of live registers for a portion of `particle_filter`, with low live register points highlighted

This matches register usage patterns in kernels. The number of live registers changes over time in a program – for example, while a complex expression is being computed, there will be many live registers to hold intermediate values, but these will be collapsed to a single value at the end. These points with fewer live registers form natural seams in the program for region boundaries. Figure 5 shows these seams in a portion of the `particle_filter` application.

It is also important that a long-latency global load and its first use are not inside the same region. If a warp were to stall on a long-latency load in the middle of a region, it would consume space in the OSU while not being able to issue any instructions. Instead, long-latency operations should happen on the edges between regions to overlap the time the register is waiting for the load with the time it is waiting for capacity in the staging unit. To achieve this, the compiler splits regions containing a load and its use.

Unlike the strands in [11], we do not allow regions to span basic block boundaries, which allows the register management to be oblivious of control flow. This does not increase data movement, since the OSU only evicts regions' output registers when more capacity is needed – if two regions from the same warp are scheduled close to each other in time, many of the input registers of the second region are often still in the OSU and are never transferred from memory. RegLess' register usage annotations are more specific than those in Zorua [63] as RegLess manages exactly which registers hold live values across region boundaries, not only how much register capacity is needed overall.

4.2 Region Creation Algorithm

RegLess' region creation algorithm is shown in Algorithm 1. The `CreateRegions` procedure starts by creating a control flow graph with regions equal to basic blocks. It then iterates through each region, determines whether it meets all constraints, and if not splits it into two regions. The first new region from the split is guaranteed to be valid, but the second must be re-examined by the algorithm.

The `IsValid` function determines whether a region is valid by checking whether the region uses few enough registers to fit in the staging unit hardware. The maximum number of registers used in the region is used to limit the amount of the staging unit one region can fill, so that one region cannot take up too large a fraction of the OSU and limit concurrency (line 18). Because the staging unit is split into multiple banks, the registers used by a region must fit

Algorithm 1 Region Creation

```
1: function CreateRegions(cfg)
2:     regions ← ∅
3:     worklist ← basic blocks in cfg
4:     while worklist is not empty do
5:         region ← worklist.pop()
6:         if not IsValid(region) then
7:             splitPc ← FindSplitPoint(region)
8:             Split region at splitPc into firstRegion and secondRegion
9:             region ← firstRegion
10:            worklist.append(secondRegion)
11:        end if
12:        regions.append(region)
13:    end while
14:    return regions
15: end function
16:
17: function IsValid(region)
18:    if region.maxLiveRegs > maximum registers per region then
19:        return false
20:    else if region.maxRegsPerBank > registers in each OSU bank then
21:        return false
22:    else if region contains a global load and its first use then
23:        return false
24:    end if
25:    return true
26: end function
27:
28: function FindSplitPoint(region)
29:    upperBound ← first PC where the first region becomes invalid
30:    lowerBound ← PC <= upperBound where the number of global
           loads and uses in both new regions is minimized
31:    lowerBound ← min(max(region.startPC + 48, lowerBound),
           upperBound)
32:    return PC such that lowerBound <= PC <= upperBound and split-
           ting at PC results in the fewest number of input and output
           registers in both new regions combined
33: end function
```

inside those banks (line 20). Finally, a global load and its first use may not be in the same region (line 22).

To determine where to split a region, the `FindSplitPoint` function identifies a window in which the split should happen. The last instruction in this window (*upperBound*) is the first PC where the first new region from the split would be invalid. The first instruction in this window (*lowerBound*) is the place that would put the region boundary between the most global loads and their first uses. The beginning of the window is adjusted to contain at least six instructions if possible, to avoid degenerately small regions. Then, the region is split at the point in this window where the split would create the least amount of input and output registers.

The annotations in Figure 6 that come from this compiler analysis are the bank usages of the input and interior registers, as well as the registers to preload.

4.3 Register Lifetime

Because both the staging unit and L1 cache have very limited capacity, it is vital that no space be consumed by dead registers. In order for the compiler to inform the hardware about when registers

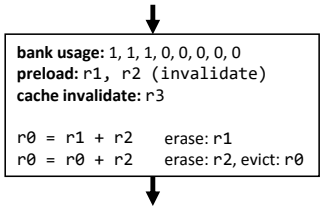

Figure 6: Compiler annotations added on regions and instructions

die, the compiler needs to take into account the two places where registers can be stored. Both interior registers and inputs and outputs can be stored in the staging unit, but only inputs and outputs can be evicted to L1. Therefore, the hardware structures in which a dead register needs to be erased depends on whether it is interior to a region or not.

Since registers with their entire lifetime within one region will only be stored in the staging unit, it is sufficient to mark the last use of the register in the region. In Figure 6, this is the *erase* annotation. Input and output registers also have a lifetime in the staging unit while a region is executing, in that there is some point in the region where they will be used for the last time in that region. These last uses are marked by the *evict* annotation in Figure 6 – note this does not mean the register must be evicted from the staging unit, only that it becomes eligible for eviction at that point.

The lifetimes of registers that live longer than one region need to be tracked so they can be erased from the L1 cache when no longer needed. These registers can either die when preloaded for the last time or when control flow eliminates the possibility of another preload. In the case that a preload is the last use of a register, the preload is set as an invalidating read, like `r2` in Figure 6. Registers known to be dead due to control flow at the beginning of a region are marked for cache invalidation, like `r3` in Figure 6.

4.4 Control Flow and Register Liveness

Finding the correct location to insert register invalidations is nontrivial problem on a GPU, because the threads in a warp can diverge for control flow. If not all lanes in a warp are active, a write to a register will only write to some parts of the register. Therefore, standard liveness analysis will produce incorrect results for GPU code, because it assumes that writing to a register kills the entire value. We call a definition that may not redefine an entire register's value a *soft definition*.

Tracking register liveness accurately is important for inserting cache invalidations in the correct place. A cache invalidation annotation deletes the entire register, not just the values for active lanes, so it is only safe to insert an invalidation when the entire register is known to be dead. Previous work [19] recognized this and described how invalidations must be inserted in a postdominator of both the definitions and uses in a live range. That is, the divergent control paths that use the register must reconverge before the invalidation. We expand on this contribution with more details about how to compute live ranges for GPU registers while accounting for control divergence.

To do so, liveness analysis must determine which definitions of a register are soft definitions. Algorithm 2 decides whether an instruction *insn* that defines a register *reg* is a soft definition, which

Algorithm 2 Identifying Soft Definitions

```
1:  procedure IsSoftDef(insn, reg)
2:      insnBB ← the BB containing insn
3:      strictDoms ← dominators(insnBB)−insnBB
4:      for all domBB in strictDoms do
5:          strictPDoms ← postdominators(domBB) −domBB
6:          if dominators(insnBB) ∩ strictPDoms ≠ ∅ then
7:              continue
8:          end if
9:          for all successorBB of domBB do
10:             if successorBB dominates insnBB then
11:                 continue
12:             else if reg is live on the edge from domBB to successorBB
                    then
13:                 return true
14:             end if
15:         end for
16:     end for
17:     return false
18: end procedure
```

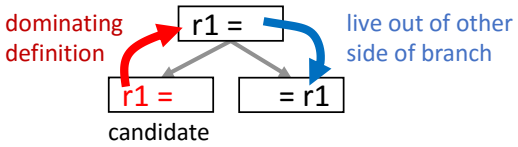

Figure 7: Determining whether a definition is soft. A soft definition of a register might not kill every thread's values.

is shown graphically in Figure 7. For a definition to be soft, there must be another definition that reaches a use with different control conditions than the candidate soft definition. Therefore, first the algorithm builds a list of the basic blocks that dominate the candidate soft definition, other than its own basic block (lines 2-3). (A basic block dominates another if control must pass through the dominator before the other basic block, and a basic block postdominates another if control must pass through the postdominator after that basic block.) Then, for each dominator, it tests whether there is a reconvergence point between the dominator and the candidate soft definition, done by testing whether there are any basic blocks that postdominate the dominator that dominate the definition (lines 6-8). This ensures that the dominating definition used is the nearest. Finally, it tests whether there is a successor with different control conditions than the candidate soft definition (lines 10-11) that uses the dominating definition (lines 12-13). If so, the candidate is a soft definition.

To compute when values die, standard dataflow analysis is used to compute live ranges, with the change that a live range does not end at a soft definition. Next, the death points of each live range are determined – either a last use or a control flow edge out of a loop. To cover the case where a register is defined but not used along a control flow path, the invalidation annotation is places in the postdominator of all the definitions and death points of the live range. Registers with a soft definition in a region are annotated for preloading, so that the values in lanes not taking the control flow path are preserved.

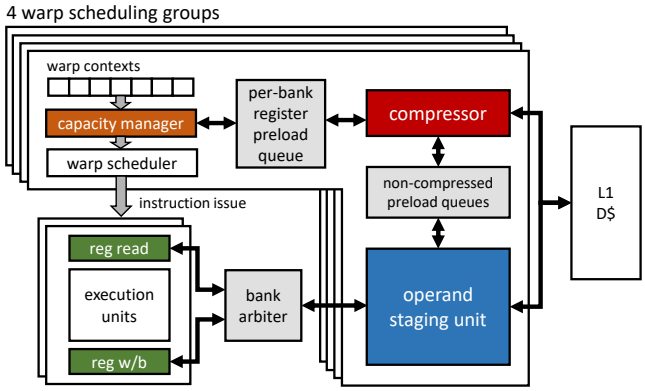

Figure 8: Block diagram of RegLess components in each SM

5 HARDWARE DESIGN

At run time, the hardware follows these compiler annotations to manage staging unit capacity. **Capacity managers (CMs)** use the register usage annotations to make allocations for warps in the staging units. The **operand staging units (OSUs)** store registers for active warps and transfers registers to and from L1 as needed to run new regions. **Compressor units** compress registers transferred to L1 to conserve capacity. Figure 8 shows how these RegLess components are integrated into an SM.

There is a separate shard of RegLess for each of the four warp schedulers in the GTX 980. That is, each of the schedulers has its own CM, OSU, and compressor unit. Multiple warp schedulers allow the GPU to easily issue multiple instructions per cycle, so making independent register scheduling decisions for each scheduler is important to keep this concurrency. No communication between shards is necessary because warps cannot read each others' registers. However, only one shard can access the L1 at a time, as the L1 cache can only accept one request per cycle.

The CMs sit in front of the warp schedulers, allocating space in its OSU for warps as they begin regions, and only allow the warp scheduler to issue instructions from warps that have their registers ready. The CMs read from a metadata store, not shown, which is filled by the decode stage. Active warps read their registers from an OSU. Before a warp can become active, it must assemble its active registers in the OSU, either from registers already in the OSU or by loading them from L1. Any unallocated OSU capacity is used to cache output registers for inactive warps in case they are inputs to another region.

Each execution unit has a corresponding register read unit that assembles the source operands from the OSU and reserves space for the destination registers for each instruction. After an instruction is finished executing, its value is written back to the OSU. Since instructions at the execution units may be from any warp in any scheduling group, an arbiter directs register reads and writes to the correct OSU.

In order to reduce the memory system throughput requirements of loading and storing registers from memory, compressors identify common patterns in register values, storing a compressed representation of a register if possible. The compressors contain a small amount of storage to cache compressed values.

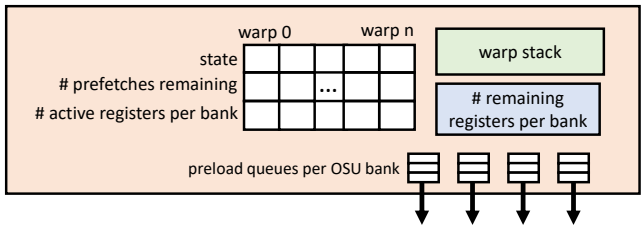

Figure 9: Capacity manager (CM) design. CMs track which warps have registers allocated in the OSU and are ready to execute instructions.

5.1 Capacity Managers (CMs)

The capacity manager is responsible for allocating OSU resources to active and preloading warps. Figure 9 shows the components of the capacity managers. Each CM contains state machines for its supervised warps that tracks whether they are in an inactive, active, or preloading state, as well as counters tracking the number of preloads and evictions to determine when the states should transition. They also maintain a list of inactive warps in the *warp stack*. The top warp in the stack is the last one to have executed, so its input registers are the most likely to already be in the OSU.

Each cycle, the CM determines whether there is enough free capacity in its OSU to activate the top warp on the stack, by comparing the registers needed by the warp's next region against a counter of free registers. If there is space in the OSU, the CM places the registers the compiler annotated to be preloaded or evicted into queues to send to the OSU banks and updates the warp stack and counters. There is a queue entry for each line in the OSU banks, so there is guaranteed to be enough queue space to insert the preloads. The OSU notifies the CM as preloads and evictions are processed, and once all of them are completed that region's warp is activated and the warp scheduler can issue instructions from that warp. The warp scheduler does not require any changes from the baseline GTO policy.

When a region has issued its last instruction, there still may be registers that have yet to be written back to the OSU. For example, if the last instruction in the region is a global load, the value may take hundreds of cycles to be written back. While it waits, any other registers that were allocated to that region can be freed for other warps, but the pending register must stay allocated. The capacity manager tracks the number of outstanding writes for a region, and keeps its state machine in a draining state until all of its registers are written. At that point, the final registers are reclaimed and the warp is deactivated and pushed onto the warp stack.

5.2 Operand Staging Units (OSUs)

The operand staging units store the register values for active and preloading warps. Each OSU is made of 8 independent banks, which are independently tracked by the CM. Each cycle, the register read and writeback units in the execution units arbitrate for access to the OSU banks of the warps and registers that they need; each bank can process either a read or a write per cycle. To read a register, the read units request a value from a bank. To write a register, the read units request an OSU entry for the future writeback, which the writeback units provide once the instruction has completed.

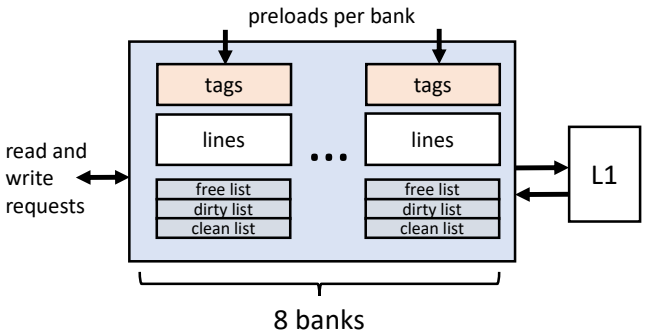

Figure 10: Operand staging unit (OSU) design. OSUs store register values and service register read and writes.

Figure 10 shows the structure of the OSUs. There are 8 banks in each OSU, with registers assigned to a bank by taking the lower 3 bits of the sum of the warp ID and register number (the compiler selects register numbers in a manner that reduces bank conflicts). Most instructions require 2 register reads and 1 write, so it is possible for each OSU to service two instructions per cycle, necessary to match the dual-issue capability of the GTX 980 schedulers. The tags in each bank store the warp ID and register ID, matching those to a 128-byte line in the data store. Each bank can complete one tag lookup per cycle, which is used when performing a register read or preload. The OSU maintains three lists of lines that are not being used by an active region: the free list tracks empty entries, the clean list tracks registers that have not changed value since being read from L1, and the dirty list tracks lines written since their last read from L1. When a register is allocated, an entry is used from the free list if possible, then from the clean list if necessary, then from the dirty list if needed, which reduces the number of writebacks needed to the L1.

5.2.1 Preloads and Allocations. Registers are allocated in the OSU either through preloads or writes to interior registers. Preloads are passed from the capacity manager for each bank in parallel. If the tag access for a bank was not used by a register read in a cycle, the bank can process the preload by checking to see if the register is present in the bank. If it is present, the register is removed from the clean and dirty lists; if it is not, the bank passes the request on to the compressor. The compressor either replies with the full register value or with a signal that the register was never compressed, in which case the OSU fetches the value from L1. Cache invalidation requests are sent through this pipeline as well, but are routed immediately to the L1 cache. For interior registers, space is allocated when a warp writes to the register for the first time.

5.2.2 Evictions. The register lifetime annotations inserted by the compiler determine when register values are no longer needed. Registers marked for invalidation are added to the free list to be recycled. Output registers marked for eviction are placed in the clean or dirty list, depending on the value of a dirty bit that is set if the register is written. When a register write is the last use of a register in a region, the OSU passes a flag to the register read unit that reserved an entry for the write. This flag is later passed with the write's value, telling the OSU to mark the register as evictable and dirty as soon as it is written.

5.2.3 Register to Memory Mapping. The memory space for registers is allocated by `cudaMalloc()`, similar to other global memory buffers. Our CUDA API detects when the first kernel is launched in an application, and makes this allocation automatically. The register base pointer is passed to hardware like a kernel parameter, and the registers are laid out in memory in order of register number, such that all the values of R0 for every warp are sequential in memory, then all the values of R1, and so on. Because different warps tend to access the same register numbers close to the same time, this minimizes cache set conflicts.

The L1 cache is by default write-through and write-evict, which would prevent dirty register values from being stored in the L1. We modify the L1 to be write-back for register values with the added optimization that the old value does not need to be fetched from memory on a write, as we guarantee the write will overwrite the entire cache line by preloading any register that may be only partially written.

5.3 Compressor

Register compression is able to reduce the amount of memory traffic required to supply the OSU. The goal of compression is to reduce both the number of accesses sent to the L1 and the space each cold register consumes, as both L1 bandwidth and capacity are scarce. Instead of needing to fetch or evict one cache line per register, many compressed registers can be stored in one line. As registers move in and out of the OSU when preloaded or evicted, a compressor matches the register value against a set of patterns and if possible moves only a compressed representation to and from the L1 cache. The compressor also contains a small cache for compressed registers.

For preloads, the compressor is on the datapath between the CM and the OSU. The register index is first matched against a bit vector which tracks whether a register is compressed. This way, the compressor does not need to bring in a line of compressed registers from the L1 only to determine whether a register is compressed. Evictions from the OSU first pass through the compressor, where the value is matched against common patterns by a compression unit. Any misses or incompressible evictions return to the OSU to be sent to the memory system.

Compression is effective due to the way kernels use registers. Previous work [31, 41] also took advantage of this with a general-purpose compression scheme, but RegLess uses a simpler scheme that matches a set of common use patterns. These patterns are constants, where all lanes of the register have the same value, stride one values, stride four values, and half-warp versions of the stride one and four patterns. For each compressed register, 8 bytes need to be stored for values for the half warp cases and 4 for the others. There are 5 compression schemes and the uncompressed state, so 3 bits per register are needed to store the state. This means that 15 compressed registers can be stored in a 128-byte cache line. Compressed lines are mapped to a separate main memory space adjacent to the uncompressed registers.

The compressor adds one extra cycle of latency for non-compressed preloads, to match against the bit vector. Compressed registers require two more cycles to match against the compressor's tags then uncompress and return the value. This added delay is small compared to the benefit of using less of the limited throughput to the L1, and preloading registers ahead of time allows this latency to be hidden. The compressor also adds similar delay when compressing registers evicted to L1, but this latency does not affect the rate warps become active.

SMs	16, 64 warps each, 4 schedulers
Warp scheduler	GTO
L1 cache	48KB, 32 MSHRs, data accesses bypassed [46]
L1 bandwidth	one request per cycle
Memory system	2 MB L2, 4 memory partitions, 224 GB/s B/W
Compressor	one read or write per cycle, 16 lines internal storage (48 per SM)

Table 1: GPGPU-sim simulation parameters

5.4 Metadata Encoding

Metadata is inserted into the instruction stream by the compiler. With 10 bits of each 64-bit instruction used for the opcode [19], 54 bits of metadata can be passed per instruction. A region starts with a flag instruction which includes the bank usage and up to 3 preloads and cache evictions; more metadata instructions for preloads and cache evictions are emitted as necessary. For every 9 instructions in a region, a metadata instruction is emitted to mark when the last uses of registers: 1 bit to determine whether an operand is a last use, and a second for whether it is an erase or invalidate flag. Some regions, especially in control-flow intensive code, have few instructions but correspondingly few preloads and invalidations, so a single-instruction encoding is used for these that can encode up to 2 preloads or invalidations and flags for up to 4 instructions.

6 EVALUATION

6.1 Methodology

RegLess was implemented in GPGPU-sim 3.2.2 [5], with the parameters in Table 1 based on the GTX 980. Register assignment was done by `ptxas` and loaded into GPGPU-sim as PtxPlus, and the compiler infrastructure used a custom framework built upon GPGPU-sim's IR. Every benchmark in the Rodinia [6] benchmark suite was used, to evaluate against many different types of GPU workloads. The simulation accounts for the performance and energy impact of the metadata inserted into the instruction stream.

We implemented RegLess and the baseline register file design (including register banks, arbitration logic for register read and write back units, and operand collectors) in Verilog and synthesized it to a 28 nm technology netlist using Synopsys' Design Compiler. Clock gating was implemented in RegLess and the baseline to reduce power consumption during periods of inactivity. Interconnect overhead was estimated by using Cadence's Encounter tool to place-and-route the designs and extract the resistance and capacitance values of the circuits. Traces from the GPGPU-sim simulations were used to stimulate the netlist running at 1GHz in order to produce power metrics. Power information for added L1, L2, and DRAM accesses came from GPUWattch [33].

We compared the register file and overall GPU energy savings against two other register file energy saving schemes. The first is Jeon et. al [19] (RFV), which reduces the size of the register file by renaming short-lived registers. Our implementation assumes a half-size register file and a negligible cost for the rename table and metadata instructions. The other technique, in Gebhart et. al [11]

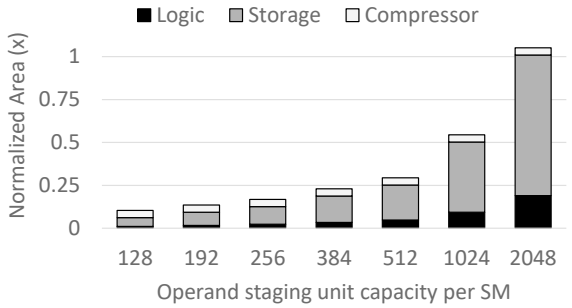

Figure 11: Area for RegLess configurations, normalized to 2048-entry baseline RF

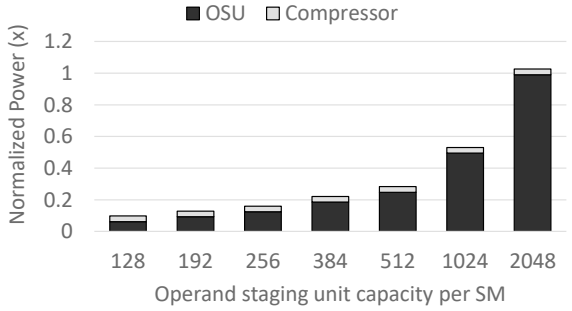

Figure 12: Combined static and average dynamic power for RegLess configurations, normalized to baseline RF

(RFH) uses a compiler technique to place registers in one of two smaller structures instead of the main register file when possible; we implemented the compiler technique and modelled the register file and added component energy in the same process technology as RegLess. We do not compare against works that repurpose unused register file space for other memory spaces, such as [12, 23], because their benefit comes through increasing occupancy or L1 capacity.

6.2 Area and Power

We evaluated multiple capacities of RegLess to find the most energy-efficient design. The area and average power of each capacity is shown in Figures 11 and 12. Both logic and storage area scales with the capacity, as more logic is needed for tags and decoding. The average power also scaled with the capacity, since more energy was required to access the larger hardware structures. Because of the added tag and compressor logic, the RegLess designs require slightly more energy and power than the baseline register file scaled to their capacity.

Although smaller capacities use less area and power, they can also affect performance if too many registers must be transferred to L1. Figure 13 shows the geometric mean total GPU energy and running time for different RegLess capacities across all Rodinia benchmarks. Small capacities like 128 registers are Pareto-optimal in terms of energy, but our goal in RegLess was no average performance loss, so we use the 512-register version in the remainder of our results as one optimal tradeoff point between performance and energy; this capacity has better worst-case performance than the 384-register version. Larger RegLess capacities see a slight speedup, which we discuss in Section 6.4.

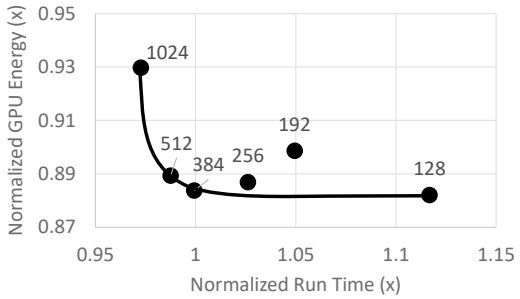

Figure 13: Run time vs. GPU energy for RegLess configurations, normalized to baseline. The line marks the Pareto frontier.

6.3 Energy Savings

RegLess significantly reduces the energy consumed both by the register structures and by the entire GPU, as shown in Figures 14 and 15. Focusing first on register structure energy, RegLess provided a 75.3% reduction, as compared to 45.2% for RFV and 62.0% for RFH; this added benefit came from reducing the amount of register storage below what was possible with the previous techniques. Because the register structures make up a significant amount of overall GPU energy, this led to a 11% overall GPU energy savings for RegLess, compared to 3.7% for RFV and 2.9% for RFH. When computing the overall GPU energy for RegLess, the cost of added L1, L2, and DRAM traffic was included. Figure 15 also shows how RegLess approaches the upper bound for GPU energy savings from reducing register file energy, 16.7%, which comes from maintaining the performance of the baseline while incurring no register file energy cost.

Compared to RFV, RegLess can maintain a register structure of half the size of even the reduced register file because of the synergy between the compiler and hardware manager. As well, some register-intensive benchmarks like `dwt2d` and `hotspot` saw performance degradation with RFV due to register pressure, as noted in their paper [19]. Compared to RFH, RegLess is able to eliminate the register file backing the compiler-managed buffer. Although RFH can save energy by accessing the large main register file significantly fewer times than the baseline, each access to that register file is more expensive than to RegLess' staging units. A two-level warp scheduler is integral to the RFH technique, which can cause performance loss relative to the baseline GTO scheduler, causing RFH to consume more energy.

6.4 Performance

Despite the much smaller register structure, RegLess is able to maintain application performance. Figure 16 shows the performance impact of RegLess on the Rodinia benchmarks relative to the baseline with a full register file, demonstrating that RegLess can match the baseline run time. Most benchmarks, such as `b+tree`, `myocyte`, and `streamcluster` saw no performance change; many of these have a small register working set that RegLess is able to easily manage. Three benchmarks (`gaussian`, `heartwall`, and `hybridsort`) saw over 5% slowdown with RegLess. `hybridsort` and `heartwall` have kernels with complex control flow structures; since registers can often not be invalidated until their last use along all paths, there are a large amount of potentially live registers that RegLess

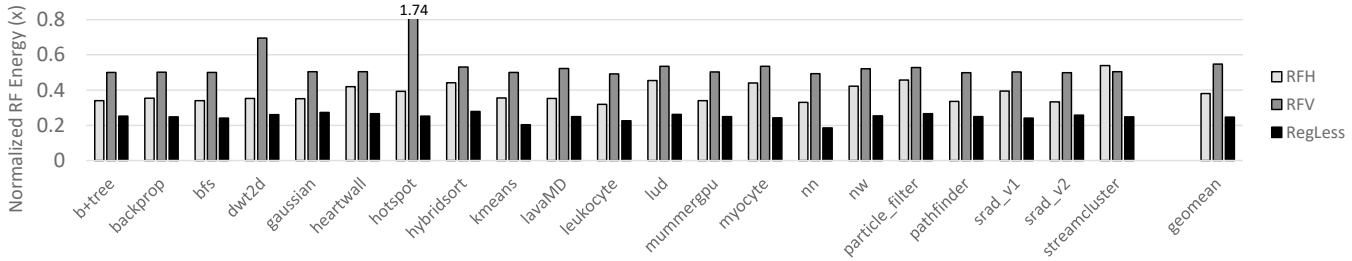

Figure 14: Register file energy for RFV [19], RFV [11], and RegLess, normalized to baseline

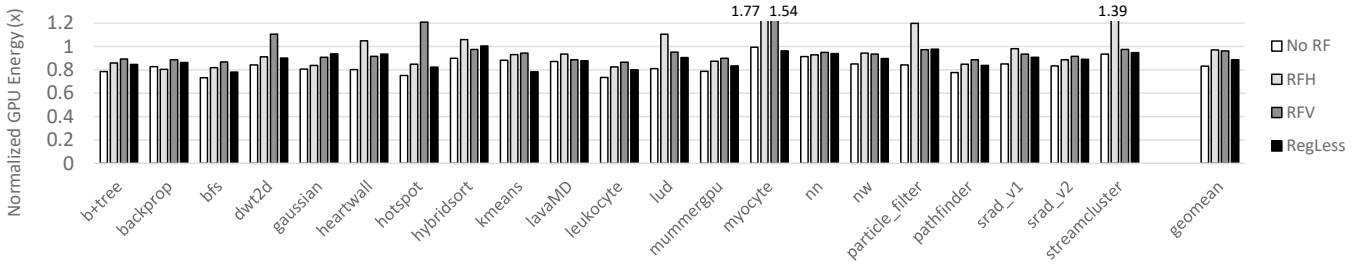

Figure 15: Normalized total GPU energy, including added instruction and memory accesses. The "No RF" entry is the upper bound for energy savings, which uses the baseline performance and a register file that consumes no energy.

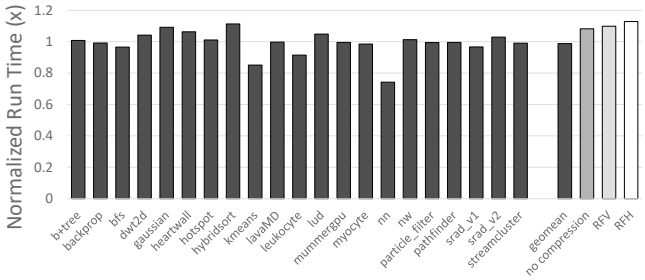

Figure 16: Run time (lower is better) for 512-register RegLess design normalized to baseline with full RF. The geomean is compared with RegLess with no compressor, RFV, and RFH.

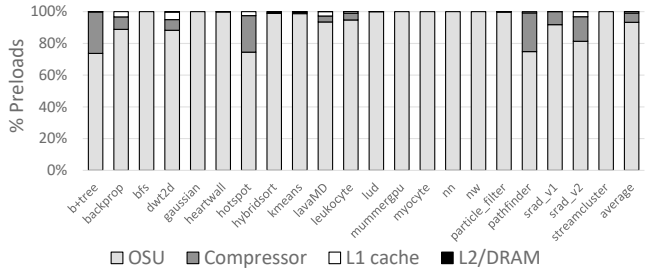

Figure 17: Location from which registers were preloaded. 0.9% of registers were preloaded from L1 and 0.013% were preloaded from L2 or DRAM.

must manage. `gaussian` has many registers live across global loads, which means that there are fewer opportunities for scheduling consecutive regions from the same warp. Other benchmarks, like `kmeans`, `leukocyte`, and `nn` saw speedup, because RegLess activates fewer warps at a time than the baseline, increasing temporal locality between memory accesses. Other register file work has seen the same effect.

Figure 16 also compares the RegLess geometric mean performance with other configurations. The first is the same size RegLess but without the compressor, which degrades performance by 10.2%. We also compare against the geometric mean performance of RFV and RFH, which are slower than RegLess due to their use of a 2-level warp scheduler. RegLess is independent of the choice of warp scheduler, allowing it to use the baseline GTO which is known to perform better than 2-level schedulers due to better memory locality [56].

6.5 Register Preload Location, L1 Bandwidth

Although the memory system is the backing store for the OSUs, Figure 17 shows that preloads very rarely need to access it. Some benchmarks, like `bfs` and `nw` never miss in the OSUs, because their register working set is small. Others, like `b+tree`, `hotspot`, and `pathfinder` use the extra capacity the compressors provide. Only an average of 0.9% of requests miss to the L1 cache and 0.013% miss to lower levels of the memory system. The only benchmarks that had a non-negligible number of added L2 accesses were `kmeans` (0.5% added requests), `hybridsort` (1.0%), and `dwt2d` (2.6%). For `dwt2d` and others, this is due to a large number of simultaneously live registers, few of which are compressible.

Figure 18 shows the average amount of L1 bandwidth consumed by transfers to the compressor and OSU per SM during the execution of each benchmark, out of the total L1 cache bandwidth of 1 request per cycle. On average, fewer than 0.02 requests per cycle were used for RegLess transfers. The benchmarks that do not miss in the OSU in Figure 17 do not consume any L1 bandwidth. Both `hybridsort` and `srad_v2` issue more stores to L1 than loads; this occurs when there are redefinitions of a register on a control path before the register is read. For `hybridsort`, conservative liveness analysis again meant that more register values had to be stored than were later read.

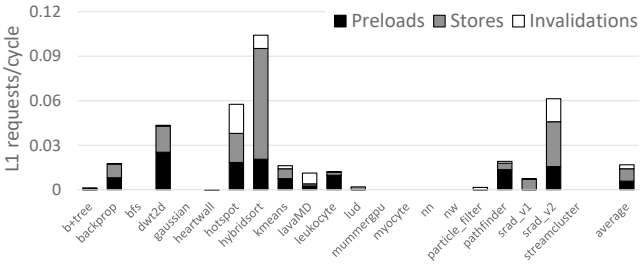

Figure 18: Average RegLess L1 requests per cycle

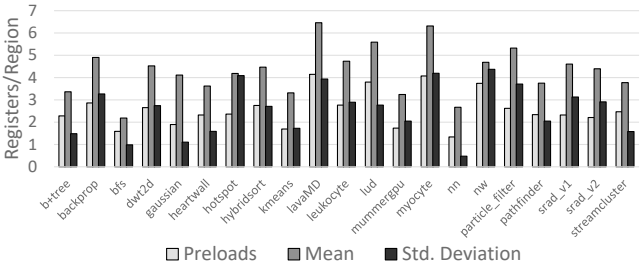

Figure 19: Average number of preloads, average number of concurrent live registers, and standard deviation of number of concurrent live registers per region

6.6 Region Sizes

Figure 19 shows the average number of input registers, average concurrent live registers, and standard deviation of concurrent live registers for each benchmark. The number of registers reserved for a region in an OSU is equal to the number of concurrent live registers in that region. Non-overlapping short-lived registers can share the same allocation, and once an input is read for the last time, its allocation can be reused by a short-lived register. Therefore, across the benchmarks, not only is the number of concurrent live registers is consistently larger than the number of input registers, but each entry made in the OSU can be reused for several registers, showing that most register lifetimes are inside a region.

The standard deviations show that the region size varies substantially within each benchmark. Since the registers in each region cannot be negative, the standard deviations show a larger variation than if that were possible. The heterogeneity in region sizes allows warps to have different-sized register allocations at different points in execution. The region creation algorithm tends to creates smaller regions in memory-intensive or control-intensive phases and larger regions when the workload is compute-intensive, leading naturally to a mixture of different region sizes. Several of the benchmarks like `dwt2d`, `hotspot`, and `myocyte` had regions with 20 or more concurrent live registers.

Table 2 shows the average number of instructions per region and the average number of cycles each region was active for each benchmark. Larger regions allow there to be more interior registers, and longer-running regions reduce the rate at which L1 transfers are made. The main factors that limit region size are control flow and the restriction that global loads and uses cannot be in the same region. Therefore, compute-intensive benchmarks like `dwt2d`, `lud`, and `nw` have the largest region sizes, whereas memory-intensive

	Insns.	Cycles
b+tree	3.7	150
backprop	6.7	323
bfs	3.3	60
dwt2d	9.5	457
gaussian	8.1	1207
heartwall	4.6	32
hotspot	6.4	75

	Insns.	Cycles
hybridsort	6.5	379
kmeans	3.9	993
lavaMD	7.5	1601
leukocyte	7.7	297
lud	16.0	816
mummergpu	6.4	240
myocyte	9.3	120

	Insns.	Cycles
nn	6.3	940
nw	10.8	78
particle_filter	10.0	20
pathfinder	4.9	72
srad_v1	9.1	350
srad_v2	6.9	323
streamcluster	4.3	16

Table 2: Average number of static instructions per region and average dynamic cycles per region

benchmarks like `bfs` have smaller region sizes. There is large variation in how long regions execute, influenced by the number of instructions in a region and how many registers active regions use. When each warp has a large OSU allocation, fewer warps will be active, so active warps will make more progress than if more warps with smaller allocations were active. Therefore, memory-intensive `bfs`, with small regions with few registers each, has a smaller execution time per region, whereas `lavaMD` with larger regions with many registers switches regions less frequently.

7 RELATED WORK

CPU virtual register files and instruction clustering: Oehmke et al. [47] created a *virtual context architecture* for CPUs that serviced registers from cache of a register space in memory. Because the amount of data in a GPU's register working set is much larger because many threads are active at the same time, our technique requires more active management of the register cache. Roth [57] describes techniques for releasing virtual registers when they are no longer needed. Architectures such as TRIPS [13, 58] and others using block-structured ISAs, described by Melvin et al. [43], have executed blocks of code similar to our regions. Work such as by Ponomarev et al. [53] have diverted short-lived values from handling like other registers. Yan et al. [68] allow short-lived values to be communicated through a CPU's forwarding network. We use regions as an overlay of a traditional ISA.

GPU register caching and RF size reduction: Vijaykumar et al. [63] oversubscribe resources, including registers, by annotating kernel phases. Our work focuses on reducing the size of hardware structures, and uses a more precise set of registers that need to be present. RegLess would be able to oversubscribe the register file without any design changes. Gebhart et al. [9] proposed a register cache in front of the main register file and a 2-level scheduling scheme to control access to the cache, to save the dynamic power of accessing the main register file. Other work by Gebhart et al. [10, 11] sorted registers at compile time into a 3-level register storage hierarchy, also to save dynamic power. The novel contribution of our work is eliminating the main register file as a level in the register hierarchy. Gebhart et al. [12] also propose sharing the same SRAM structures between registers, shared memory, and L1 cache. Jeon et al. [19] allow new values to replace other warps' dead values in the register file, allowing the size of the register file to be reduced. By removing the main register file and caching the active set, our technique reduces the register size to the minimum needed to maintain performance.

Compiler-assisted GPU scheduling: Park et al. [49] use compiler annotations so the warp scheduler can prioritize warps with

that will soon issue a load. Wu et al. [66] expose hardware scheduling decisions on GPUs to programmers. Xie et al. [67] use a compiler to make optimal register allocation and thread throttling decisions. We add a layer of scheduling that makes dynamic decisions based on static analysis. Hsieh et al. [17] use compiler analysis to determine offload candidates for near-data processing. Li et al. [36] use compiler analysis to place data in different on-chip memory resources.

Resource-aware GPU scheduling: Jog et al. [26] classify warps into short and long latency to determine memory scheduling policy. Jog et al. coordinate warp scheduling with DRAM bank-level parallelism [24] and prefetching [25]. Li et al. [37] allocate cache space to a set of prioritized warps. Narasiman et al. [45] describe two-level scheduling to allow for larger warp sizes. Pichai et al. [52] show the need to coordinate warp scheduling and MMU designs. Pai et al. [48] use elastic kernels in order to better utilize registers. Gregg et al. [16] merge kernels to increase register utilization. Lee et al. [32] coordinate warp priority and access to cache resources, Liu et al. [40] prioritize warps to reduce time waiting for barriers. Kayiran et al. [27] adjust TLP for highest performance. Rogers et al. [55] use variable warp sizing and warp ganging to decrease the impact of memory divergence. Ausavarungnirun et al. [4] change cache and memory controller policies based on warp divergence.

Divergence-aware compiler techniques: ElTantawy et al. [8] track register dependencies for control divergent threads separately in hardware, and use compiler analysis [7] to analyze control divergence to eliminate deadlocks. Rhu et al. [54] analyze divergence patterns to allow for better SIMD lane permutation. Anantpur et al. [3] transforms control divergence using linearization. Jablin et al. [18] use traces for instruction scheduling on GPUs.

Value compression and scalarization: Lee et al. [31] compress register values using base-delta-immediate encoding introduced by Pekhimenko et al. [51], which reduces the number of register file banks needed to load and store registers. Gilani et al. [14] propose a GPU architecture with scalar units and 16-bit register reads. Abdel-Majeed et al. [2] use the redundant computations done between lanes for error detection. Kim et al. [30] exploit value structure using an affine functional unit. Stephenson et al. [59] show that a large fraction of register writes are constant across warps and threads. Pekhimenko et al. [50] compress data over the GPU interconnect while minimizing the number of toggles. Vijaykumar et al. [64] propose using excess GPU computation resources for memory compression. Keckler et al. [29] propose temporal SIMT, where scalar computations do not need to be computed by all threads.

Register file implementation: Abdel-Majeed et al. [1] reduce register file dynamic and leakage power by adding a drowsy state to the storage circuits and only reading register values for active lanes in a warp. Jing et al. [20] propose register file bank scheduling techniques that reduce bank conflicts. Namaki-Shoushtari et al. [44] power gate unused register file banks. Other work by Jing et al. [22] implemented the register file using eDRAM instead of SRAM and proposed refreshing the DRAM during bank bubbles [21]. Mao et al. [42] and Wang et al. [65] implement a register file using racetrack memory, and Goswami et al. [15] implement it using resistive memory. Tan et al. [61] implement the GPU register file using STT-RAM for energy savings, and Yu et al. [69] implement it with an SRAM-DRAM hybrid memory. Tan et al. [60] develop a method for classifying registers as fast or slow due to process variation, and Liang et al. [39] introduce a variable-latency register file to mitigate process variation. Li et al. [38] implement register files using a hybrid CMOS-TFET process. Our design because of its small size can be implemented using conventional techniques.

Register file voltage: Kayiran et al. [28] tune down performance of GPU register file and operand collector components to save energy. Tan et al. [62] reduce GPU register file energy with aggressive voltage reduction. Leng et al. [34, 35] throttle the register file when it causes voltage droop to reduce the GPU voltage guardband.

8 CONCLUSION

The register file is one of the structures on a GPU that consumes the most power. Our technique, RegLess, can replace the register file with a smaller staging unit by actively managing the contents at run time with the help of compiler annotations. The compiler divides the kernel into regions and annotates input register and the points where register values are used for the last time. At run time, the hardware allocates capacity in the staging unit just in time for a region to begin execution. Short-lived registers spend their entire lifetime inside one region's allocation. Longer-lived registers can be evicted to memory, so the capacity manager must anticipate they will be used in order to load them before a region becomes eligible to execute. When transferred to the L1, a compressor can reduce the amount of storage needed for a register. Using RegLess instead of a full register file reduced register access energy by 75% and total GPU energy by 11%.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers and the members of the CCCP research group for their comments and feedback. This work was supported by ARM Ltd. and National Science Foundation grants XPS-1438996 and SHF-1527301.

REFERENCES

[1] 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.

[2] M. Abdel-Majeed, W. Dweik, H. Jeon, and M. Annavaram, "Warped-re: Low-cost error detection and correction in gpus," in *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks.* IEEE, 2015, pp. 331–342.

[3] J. Anantpur and R. Govindarajan, "Taming control divergence in gpus through control flow linearization," in *International Conference on Compiler Construction.* Springer, 2014, pp. 133–153.

[4] 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).* IEEE, 2015, pp. 25–38.

[5] 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.

[6] 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.

[7] A. ElTantawy and T. M. Aamodt, "Mimd synchronization on simt architectures," in *Proceedings of the 49th annual IEEE/ACM International Symposium on Microarchitecture*, 2016.

[8] A. ElTantawy, J. W. Ma, M. O'Connor, and T. M. Aamodt, "A scalable multi-path microarchitecture for efficient gpu control flow," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA).* IEEE, 2014, pp. 248–259.

[9] 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 *ACM SIGARCH Computer Architecture News*, vol. 39, no. 3. ACM, 2011, pp. 235–246.

[10] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron, "A hierarchical thread scheduler and register file for energy-efficient throughput processors," *ACM Transactions on Computer Systems (TOCS)*, vol. 30, no. 2, p. 8, 2012.

[11] M. Gebhart, S. W. Keckler, and W. J. Dally, "A compile-time managed multi-level register file hierarchy," in *Proceedings of the 44th annual IEEE/ACM international symposium on microarchitecture*. ACM, 2011, pp. 465–476.

[12] 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*. IEEE Computer Society, 2012, pp. 96–106.

[13] M. Gebhart, B. A. Maher, K. E. Coons, J. Diamond, P. Gratz, M. Marino, N. Ranganathan, B. Robatmili, A. Smith, J. Burrill, S. W. Keckler, D. Burger, and K. S. McKinley, "An evaluation of the trips computer system," in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIV, 2009, pp. 1–12.

[14] 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.

[15] N. Goswami, B. Cao, and T. Li, "Power-performance co-optimization of throughput core architecture using resistive memory," in *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*. IEEE, 2013, pp. 342–353.

[16] C. Gregg, J. Dorn, K. Hazelwood, and K. Skadron, "Fine-grained resource sharing for concurrent gpgpu kernels," in *Presented as part of the 4th USENIX Workshop on Hot Topics in Parallelism*, 2012.

[17] K. Hsieh, E. Ebrahimi, G. Kim, N. Chatterjee, M. OâĂŹConnor, N. Vijaykumar, O. Mutlu, and S. W. Keckler, "Transparent offloading and mapping (tom): Enabling programmer-transparent near-data processing in gpu systems," in *Proceedings of the 43rd Annual International Symposium on Computer Architecture*, 2016, pp. 204–216.

[18] J. A. Jablin, T. B. Jablin, O. Mutlu, and M. Herlihy, "Warp-aware trace scheduling for gpus," in *Proceedings of the 23rd international conference on Parallel architectures and compilation*. ACM, 2014, pp. 163–174.

[19] 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.

[20] N. Jing, S. Chen, S. Jiang, L. Jiang, C. Li, and X. Liang, "Bank stealing for conflict mitigation in gpgpu register file," in *Low Power Electronics and Design (ISLPED), 2015 IEEE/ACM International Symposium on*. IEEE, 2015, pp. 55–60.

[21] N. Jing, H. Liu, Y. Lu, and X. Liang, "Compiler assisted dynamic register file in gpgpu," in *Proceedings of the 2013 International Symposium on Low Power Electronics and Design*. IEEE Press, 2013, pp. 3–8.

[22] N. Jing, Y. Shen, Y. Lu, S. Ganapathy, Z. Mao, M. Guo, R. Canal, and X. Liang, "An energy-efficient and scalable edram-based register file architecture for gpgpu," in *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3. ACM, 2013, pp. 344–355.

[23] N. Jing, J. Wang, F. Fan, W. Yu, L. Jiang, C. Li, and X. Liang, "Cache-emulated register file: An integrated on-chip memory architecture for high performance gpgpus," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 2016, pp. 1–12.

[24] 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 *ACM SIGPLAN Notices*, vol. 48, no. 4. ACM, 2013, pp. 395–406.

[25] A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "Orchestrated scheduling and prefetching for gpgpus," in *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3. ACM, 2013, pp. 332–343.

[26] A. Jog, O. Kayiran, A. Pattnaik, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "Exploiting core-criticality for enhanced gpu performance," in *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science*. ACM, 2016, pp. 351–363.

[27] 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*. IEEE Press, 2013, pp. 157–166.

[28] O. Kayıran, 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 *2016 International Conference on Parallel Architecture and Compilation (PACT)*, 2016.

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

[30] J. Kim, C. Torng, S. Srinath, D. Lockhart, and C. Batten, "Microarchitectural mechanisms to exploit value structure in simt architectures," in *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3. ACM, 2013, pp. 130–141.

[31] 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*. ACM, 2015, pp. 502–514.

[32] S.-Y. Lee, A. Arunkumar, and C.-J. Wu, "Cawa: coordinated warp scheduling and cache prioritization for critical warp acceleration of gpgpu workloads," in *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3. ACM, 2015, pp. 515–527.

[33] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "Gpuwattch: enabling energy optimizations in gpgpus," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3, pp. 487–498, 2013.

[34] J. Leng, Y. Zu, and V. J. Reddi, "Gpu voltage noise: Characterization and hierarchical smoothing of spatial and temporal voltage noise interference in gpu architectures," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2015, pp. 161–173.

[35] J. Leng, Y. Zu, M. Rhu, M. Gupta, and V. J. Reddi, "Gpuvolt: Modeling and characterizing voltage noise in gpu architectures," in *Proceedings of the 2014 international symposium on Low power electronics and design*. ACM, 2014, pp. 141–146.

[36] C. Li, Y. Yang, Z. Lin, and H. Zhou, "Automatic data placement into gpu on-chip memory resources," in *Code Generation and Optimization (CGO), 2015 IEEE/ACM International Symposium on*. IEEE, 2015, pp. 23–33.

[37] D. Li, M. Rhu, D. R. Johnson, M. O'Connor, M. Erez, D. Burger, D. S. Fussell, and S. W. Redder, "Priority-based cache allocation in throughput processors," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2015, pp. 89–100.

[38] Z. Li, J. Tan, and X. Fu, "Hybrid cmos-tfet based register files for energy-efficient gpgpus," in *Quality Electronic Design (ISQED), 2013 14th International Symposium on*. IEEE, 2013, pp. 112–119.

[39] X. Liang and D. Brooks, "Mitigating the impact of process variations on processor register files and execution units," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2006, pp. 504–514.

[40] Y. Liu, Z. Yu, L. Eeckhout, V. J. Reddi, Y. Luo, X. Wang, Z. Wang, and C. Xu, "Barrier-aware warp scheduling for throughput processors," in *Proceedings of the 2016 International Conference on Supercomputing*. ACM, 2016, p. 42.

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

[42] M. Mao, W. Wen, Y. Zhang, Y. Chen, and H. Li, "Exploration of gpgpu register file architecture using domain-wall-shift-write based racetrack memory," in *Design Automation Conference (DAC), 2014 51st ACM/EDAC/IEEE*. IEEE, 2014, pp. 1–6.

[43] S. Melvin and Y. Patt, "Enhancing instruction scheduling with a block-structured isa," *International Journal of Parallel Programming*, vol. 23, no. 3, pp. 221–243, 1995.

[44] M. Namaki-Shoushtari, A. Rahimi, N. Dutt, P. Gupta, and R. K. Gupta, "Argo: aging-aware gpgpu register file allocation," in *Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. IEEE Press, 2013, p. 30.

[45] 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*. ACM, 2011, pp. 308–317.

[46] Nvidia, "Nvidia cuda programming guide," http://docs.nvidia.com/cuda/cuda-c-programming-guide, accessed: August 2017.

[47] D. W. Oehmke, N. L. Binkert, T. Mudge, and S. K. Reinhardt, "How to fake 1000 registers," in *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2005, pp. 7–18.

[48] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan, "Improving gpgpu concurrency with elastic kernels," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13. ACM, 2013, pp. 407–418.

[49] J. J. K. Park, Y. Park, and S. Mahlke, "Elf: Maximizing memory-level parallelism for gpus with coordinated warp and fetch scheduling," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2015, p. 18.

[50] G. Pekhimenko, E. Bolotin, N. Vijaykumar, O. Mutlu, T. C. Mowry, and S. W. Keckler, "A case for toggle-aware compression for gpu systems," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2016, pp. 188–200.

[51] G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Base-delta-immediate compression: practical data compression for on-chip caches," in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*. ACM, 2012, pp. 377–388.

[52] B. Pichai, L. Hsu, and A. Bhattacharjee, "Architectural support for address translation on gpus: designing memory management units for cpu/gpus with unified address spaces," in *ACM SIGARCH Computer Architecture News*, vol. 42, no. 1.

ACM, 2014, pp. 743–758.

[53] D. Ponomarev, G. Kucuk, O. Ergin, and K. Ghose, "Reducing datapath energy through the isolation of short-lived operands," in *Parallel Architectures and Compilation Techniques, 2003. PACT 2003. Proceedings. 12th International Conference on.* IEEE, 2003, pp. 258–268.

[54] M. Rhu and M. Erez, "Maximizing simd resource utilization in gpgpus with simd lane permutation," in *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3. ACM, 2013, pp. 356–367.

[55] T. G. Rogers, D. R. Johnson, M. O'Connor, and S. W. Keckler, "A variable warp size architecture," in *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3. ACM, 2015, pp. 489–501.

[56] 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.* IEEE Computer Society, 2012, pp. 72–83.

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

[58] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore, "Exploiting ilp, tlp, and dlp with the polymorphous trips architecture," in *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on.* IEEE, 2003, pp. 422–433.

[59] M. Stephenson, S. K. Sastry Hari, Y. Lee, E. Ebrahimi, D. R. Johnson, D. Nellans, M. O'Connor, and S. W. Keckler, "Flexible software profiling of gpu architectures," in *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3. ACM, 2015, pp. 185–197.

[60] J. Tan and X. Fu, "Mitigating the susceptibility of gpgpus register file to process variations," in *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International.* IEEE, 2015, pp. 969–978.

[61] J. Tan, Z. Li, and X. Fu, "Soft-error reliability and power co-optimization for gpgpus register file using resistive memory," in *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE).* IEEE, 2015, pp. 369–374.

[62] J. Tan, S. L. Song, K. Yan, X. Fu, A. Marquez, and D. Kerbyson, "Combating the reliability challenge of gpu register file at low supply voltage," in *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation.* ACM, 2016, pp. 3–15.

[63] 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 *Proceedings of the 49th annual IEEE/ACM International Symposium on Microarchitecture*, 2016.

[64] N. Vijaykumar, G. Pekhimenko, A. Jog, A. Bhowmick, R. Ausavarungnirun, C. Das, M. Kandemir, T. C. Mowry, and O. Mutlu, "A case for core-assisted bottleneck acceleration in gpus: enabling flexible data compression with assist warps," in *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3. ACM, 2015, pp. 41–53.

[65] S. Wang, Y. Liang, C. Zhang, X. Xie, G. Sun, Y. Liu, Y. Wang, and X. Li, "Performance-centric register file design for gpus using racetrack memory," in *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC).* IEEE, 2016, pp. 25–30.

[66] B. Wu, G. Chen, D. Li, X. Shen, and J. Vetter, "Enabling and exploiting flexible task assignment on gpu through sm-centric program transformations," in *Proceedings of the 29th ACM on International Conference on Supercomputing.* ACM, 2015, pp. 119–130.

[67] X. Xie, Y. Liang, X. Li, Y. Wu, G. Sun, T. Wang, and D. Fan, "Enabling coordinated register allocation and thread-level parallelism optimization for gpus," in *Proceedings of the 48th International Symposium on Microarchitecture.* ACM, 2015, pp. 395–406.

[68] J. Yan and W. Zhang, "Exploiting virtual registers to reduce pressure on real registers," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 4, no. 4, p. 3, 2008.

[69] W.-k. S. Yu, R. Huang, S. Q. Xu, S.-E. Wang, E. Kan, and G. E. Suh, "Sram-dram hybrid memory with applications to efficient register files in fine-grained multi-threading," in *ACM SIGARCH Computer Architecture News*, vol. 39, no. 3. ACM, 2011, pp. 247–258.