Scratch That (But Cache This): A Hybrid Register Cache / Scratchpad for GPUs

Jonathan Bailey, *Student Member, IEEE,* John Kloosterman, and Scott Mahlke, *Fellow, IEEE*

***Abstract*—Graphics processing units (GPUs) are throughput-oriented architectures that implement massive multi-threading. Large, power-hungry register files are required in GPUs to support the simultaneous execution of thousands of threads on the hardware. Prior work proposed reducing register access energy by adding a small register cache to the GPU. The cache stores recently referenced registers and services subsequent accesses to these registers, reducing accesses to the main register file. Later work obtained further energy savings by replacing this cache with a compiler-managed scratchpad. We note that registers are allocated to the cache dynamically and reactively whereas registers are allocated to the scratchpad statically and proactively. Our insight is that these allocation schemes are complimentary because the cache leverages runtime information unavailable to the compiler and the scratchpad leverages compile time information unavailable to the cache. Further, there exist register access patterns that are easily captured by one structure but for which the other structure is ineffective. Instead of implementing either a register cache or scratchpad alone, we propose dividing temporary register storage capacity between a cache and a scratchpad in order to capture a broader range of register accesses. Given 12 KB of storage per streaming multiprocessor, our hybrid design reduces register energy to 38.7% of the baseline, compared to 47.9% for a register cache and 47.1% for a register scratchpad.**

***Index Terms*—compiler, energy efficiency, GPU, microarchitecture, register file**

I. Introduction

Graphics processing units are specialized accelerators that target massively multi-threaded tasks. Such tasks are complete only when all threads are complete. Therefore, GPU designs prioritize total throughput rather than individual thread latency. Power- and area-hungry features such as out-of-order execution and speculation that target thread latency are unnecessary. Instead, GPUs maintain high throughput by hiding stalls in one thread with the execution of other threads. This enables them to deliver much higher energy-efficiency than traditional CPUs.

Thousands of threads may execute on a GPU, with their instructions interleaved on a cycle-by-cycle basis. All of the state belonging to these threads must be kept resident in hardware to support this fine-grained multi-threading. In particular,

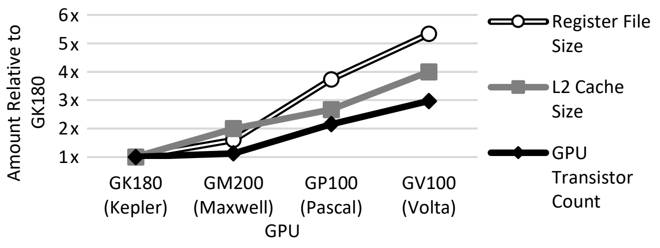


Fig. 1. Growth of resources over the four most recent Nvidia GPU generations [2]

GPUs require massive register files to store the register state of executing threads. The Nvidia GV100 [2], a member of Nvidia's most recent generation of GPUs, provisions 20 MB of register file storage. By comparison, the GV100 provides less than a third of this amount of storage (6 MB) for the L2 cache.

Because GPU register files are so large, the energy required to access (read or write) a register is high. This is despite the fact that modern GPU register files are heavily banked. For instance, prior work by Leng. et al. [3] estimated that the register file consumes an average of 13.4% of the power drawn by an Nvidia GTX 480. Further, Figure 1 shows how various resources have scaled over the past several generations of Nvidia GPUs. The expansion of register file capacity has outpaced the growth of L2 cache capacity as well as the growth of the total GPU transistor count. Therefore, it is probable that the share of GPU power consumed by the register file will remain substantial in the future. Minimizing this energy overhead will become more important as GPUs are increasingly deployed in energy-sensitive mobile and data-center environments.

As is the case for data loads and stores, register accesses exhibit temporal locality. This simply means that if an executing thread accesses a register, the thread is likely to access the same register again in the near future. Prior work by Gebhart et al. [4] leveraged this behavior by augmenting the GPU with a register cache (RC). An RC is a small structure that stores recently accessed registers and services subsequent accesses to them. Temporal locality implies that the cache will often intercept register accesses in this way. However, in the case that a thread accesses a register that is not stored in the cache, the access may still be serviced by the main register file (MRF), which is unaltered from the baseline design. The cache is smaller than the MRF and therefore requires less energy to write to and read from. Thus, an RC can elide MRF accesses

This article was presented in the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES) 2018 and appears as part of the ESWEEK-TCAD special issue. The published version of this paper [1] is available at https://ieeexplore.ieee.org/document/8413125. This work was supported by Arm Ltd and the National Science Foundation under grants XPS-1438996 and SHF-1527301.

J. Bailey, J. Kloosterman, and S. Mahlke are with the Electrical Engineering and Computer Science Department, University of Michigan, Ann Arbor, MI 48109 USA (email: jbaile@umich.edu; jklooste@umich.edu; mahlke@umich.edu)

and reduce the register energy consumed by a GPU.

Gebhart et al. [5] found that further energy savings are possible with a register scratchpad (RSP). An RC snoops the normal stream of register reads and writes, dynamically allocating registers in hardware as they are referenced. In contrast, it is the compiler's responsibility to analyze a kernel and identify the most advantageous registers to allocate to the RSP. A distinct segment of the register name space is mapped to the RSP. When the compiler allocates a register, instructions that refer to this register are modified to refer to a scratchpad-mapped register. Whereas the cache makes allocations reactively, the compiler is able to select RSP allocations proactively because it can inspect every instruction in the kernel. In other words, RC allocations may only respond to past behavior while RSP allocations may account for ”future” behavior as well.

While the proactive allocation strategy of the scratchpad approach gives it an advantage over the reactive cache as shown in prior work, the cache also has unique strengths that the scratchpad lacks, which we describe in this work. Scratchpad allocation decisions can only incorporate static information because these allocations are made at compile time, whereas the cache naturally responds to dynamic behavior. As a result, each technique is better able to capture some types of register behavior than the other. We observe that a scratchpad generally captures more register accesses than a cache for code consisting largely of sequential arithmetic operations whereas a cache generally captures more accesses than a scratchpad for code featuring many dependencies on long latency operations or complex control flow.

Our insight is that the dynamic, reactive approach of the RC and the static, proactive approach of the RSP are complementary. Rather than incorporating only one structure or the other into a GPU, we propose adding a hybrid register cache / scratchpad. In this hybrid design, a register hierarchy is formed in which the RSP is backed by the RC, which in turn is backed by the MRF. The first-level RSP leverages compile-time information to proactively allocate registers. In cases where the compiler fails to allocate a register to the scratchpad because of compiler limitations or lack of information about runtime behavior, the RC's dynamic allocation strategy is often able to capture accesses to the register instead. This allows the hybrid design to elide a broader range of register writes and reads than a cache-only or scratchpad-only design. Further, a single kernel or segment of code often includes register accesses that one structure can capture but that the other cannot. Our hybrid design benefits from a synergy between the RC and RSP components in these cases that allows it to capture more register accesses than either by itself.

Our contributions in this work include:

- Evaluating the relative strengths and weaknesses of RCs and RSPs.
- Identifying an opportunity to improve register energy efficiency with a hybrid register cache / scratchpad.
- Analyzing the characteristics that make a kernel amenable to energy savings with an RC, RSP, or hybrid design.
- Proposing a hierarchical organization for the RSP, RC, and MRF within the hybrid design and investigating the relationship between the RC and RSP subcomponents.
- Demonstrating that our hybrid design reduces register energy to 38.7% of the baseline and total GPU energy to 92.8% of the baseline, saving more energy than either a cache-only or scratchpad-only design.

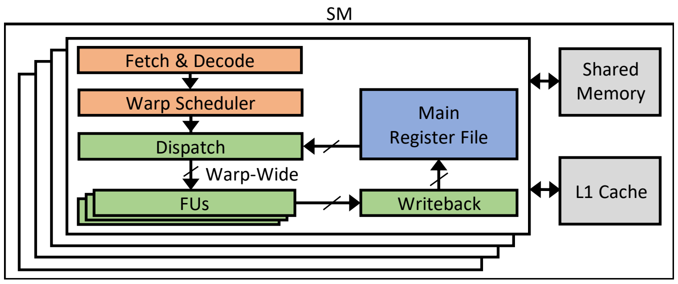


Fig. 2. Diagram of a GPU streaming multiprocessor

II. Background and Motivation

A. GPU Architecture and SIMT Execution

In this work, we consider a GPU model based on the Nvidia GTX 980 [6]. This GPU contains 16 streaming multiprocessors (SMs), partitions that have dedicated resources such as shared memory and an L1 cache. Further, each SM includes 4 warp schedulers, each of which issues instructions to private functional units. A diagram of an SM is shown in Figure 2.

Threads in a GPU kernel execute the same code and often follow the same control flow path. To leverage this redundancy, threads are collected into groups of 32 called warps. A warp is assigned to one scheduler for its entire lifetime and serves as the basic unit of scheduling and execution in the pipeline. In order to amortize control logic overheads, a warp fetches and decodes scalar instructions but executes these instructions for multiple threads simultaneously with vector functional units. This is similar to the single instruction, multiple data (SIMD) model, but threads still maintain independent execution state. In the uncommon case that threads in a warp take different control flow paths, each path is executed sequentially until a reconvergence point is reached. This execution model is called single instruction, multiple thread (SIMT) and we refer to this ordering of basic block execution as SIMT execution order.

A warp scheduler is responsible for orchestrating the execution of multiple, simultaneously running warps. In the case of the GTX 980, up to 16 warps may be assigned to a scheduler concurrently. Each cycle, the scheduler determines which warps are ready to issue instructions and arbitrates among them for access to the pipeline. This mechanism allows many warps to share the same execution hardware. Further, when a warp stalls, the scheduler can maintain high throughput and hardware utilization by issuing instructions from other warps instead of waiting for this stall to resolve. A two-level scheduler, as proposed in [4], divides warps into a small set of active warps that may issue instructions and a larger set of pending warps that may not until moved into the active set. In prior register management techniques [4], [5], [7], [8], a two-level scheduler has been effective at reducing the size of the short-term working set of warp registers seen by the

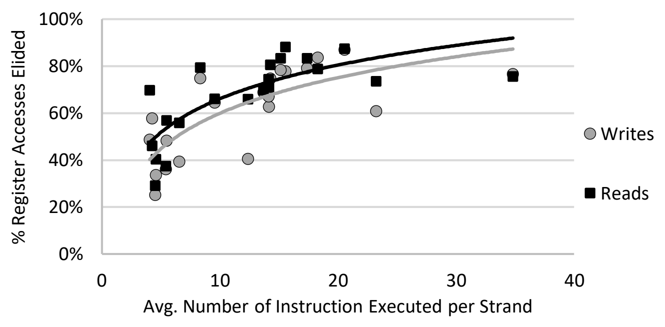


Fig. 3. Percent of register accesses elided by an RSP with 12 KB / SM capacity versus dynamic instructions per strand for all Rodinia benchmarks. Frequent strand boundaries reduce the number of register accesses the RSP captures.

hardware. We therefore employ a two-level scheduler with our hybrid design.

B. *Advantages of a Register Scratchpad*

A cache allocates registers as they are accessed. When there is contention for RC capacity, the least recently used (LRU) strategy dictates which registers to evict. As a result, the contents of an RC reflect the recent history of register accesses. This approach leverages temporal locality, which suggests that future register accesses will be similar to past accesses. However, temporal locality only offers a description of typical behavior, not a precise forecast of future behavior in a particular instance. The cache's strategy therefore constitutes a heuristic approach. This leads to suboptimal allocation and eviction decisions in some cases, reducing RC effectiveness in two ways. First, because an RC has finite storage space, allocating a register that will not be frequently accessed in the future may require evicting a register that will be frequently accessed. Second, the cache may incur extra energy overhead by unnecessarily moving data between the RC and MRF.

An ideal RC would have precise knowledge of future register accesses. With this foresight, the energy savings achieved by each possible allocation could be perfectly predicted. The cache could then optimally ration its finite storage, selecting the set of cache allocations such that total register energy is minimized.

Unfortunately, an ideal cache with perfect future knowledge cannot be implemented. However, an RSP offers much of the same benefit because allocations are selected by the compiler, which has substantial insight into future accesses. This insight is possible because the entire kernel is visible to the compiler and there is no aliasing for register accesses as there is for memory accesses. The compiler is therefore able to anticipate future accesses and proactively select scratchpad allocations whereas the cache performs allocations reactively based on past accesses. An RSP also does not require tag stores and comparisons like a cache, so RSP accesses consume less energy than RC accesses. Prior work has demonstrated that these advantages allow an RSP to achieve greater energy savings than an RC [5].

C. *Advantages of a Register Cache*

Although the compiler has insight into future register accesses, this insight is neither perfect nor complete. The compiler's knowledge is incomplete because scratchpad analysis divides the kernel into regions and only allocates registers within these regions. The compiler's knowledge is imperfect because it does not include knowledge of dynamic behavior. As a result, opportunities exist for an RC to capture accesses that an RSP does not because RC allocations are made in response to continuously observed dynamic behavior.

1) Boundaries Between Strands: While the compiler can inspect all instructions in a kernel, it can only map registers to the scratchpad within limited regions of code. Splitting the code into regions for compiler analysis of register usage is a common feature of prior work. For instance, two regions may be divided for all types of control flow [9], for instances of control flow divergence [10], at backwards branches and their targets [5], or at function calls and returns [8]. Further restrictions may be imposed on regions, such as requiring that a region have only one entry point [8], disallowing regions within which execution may stall due to unpredictable memory latencies [5], [9], or disallowing regions that require more than a certain amount of a hardware resource [8], [9]. While the details differ in each of these cases, dividing the code into regions is generally done because the dynamic behavior of executing code is not known at compile time or to simplify compilation.

In this work, we adopt the term "strand" from [5] to refer to the regions of code that the compiler inspects. The complete procedure for identifying strands is described in Section III-B, but for now we note that, among other restrictions, strands cannot contain dependencies on long-latency operations except for the first instruction in the strand and cannot span backwards branches or their targets. Allocations can only be made within a single strand and register values cannot be communicated directly through the scratchpad from producers in one strand to consumers in another. Strand boundaries therefore limit the ability of the scratchpad to reduce register energy. Figure 3 plots the percent of register writes and reads elided by a scratchpad versus the average number of instructions dynamically executed per strand for each benchmark in Rodinia [11]. These results show that shorter intervals between strand boundaries diminish the scratchpad's effectiveness. In contrast, the compiler limitations that necessitate strand boundaries are irrelevant to the cache because the cache does not depend on the compiler to make allocations.

2) Dynamic Control Flow: The ability of the compiler to effectively allocate registers to the scratchpad is also inhibited because the compiler makes RSP allocations statically. As described previously, a distinct segment of the register name space is mapped to the RSP. When the compiler allocates a register to the RSP, the instructions that reference this register are modified to reference the scratchpad-mapped register instead. Therefore, RSP allocations are fixed at compile time and cannot adapt to dynamic behavior.

The compiler is not always certain about which instructions will be executed dynamically due to control flow that cannot be predicted statically. In such cases, it is not known

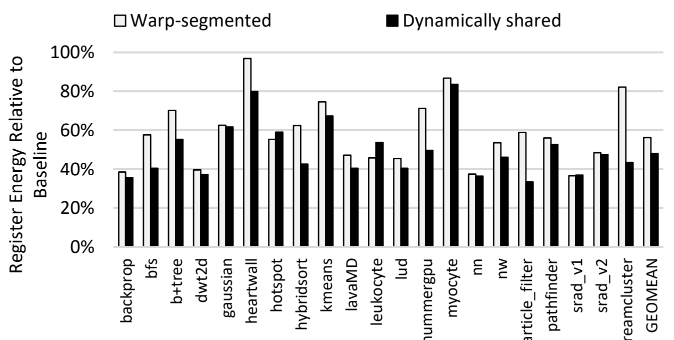
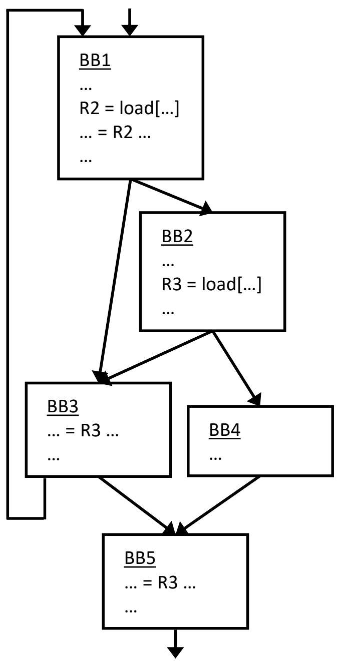
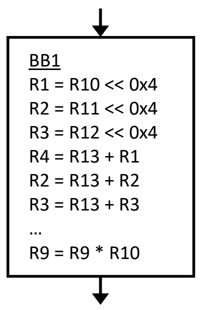


Fig. 4. Register energy for a cache that evenly divides capacity between active warps and for a cache where warps dynamically share capacity. Dynamically sharing capacity allows the storage to be used more effectively, saving energy.



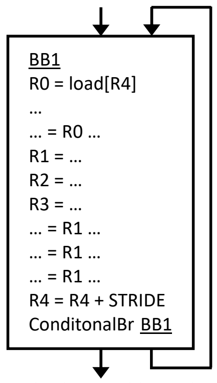
```
BB1
...
R2 = load[...]
... = R2 ...
...

BB2
...
R3 = load[...]
...

BB3
... = R3 ...
...

BB4
...

BB5
... = R3 ...
...
```

(a) Cache-amenable code. Complex control flow and long-latency dependencies impair RSP effectiveness, but an RC can respond to the dynamic behavior of these features.



```
BB1
R1 = R10 << 0x4
R2 = R11 << 0x4
R3 = R12 << 0x4
R4 = R13 + R1
R2 = R13 + R2
R3 = R13 + R3
...
R9 = R9 * R10
```

(b) Scratchpad-amenable code. The compiler can effectively anticipate register accesses for sequential arithmetic instructions and make RSP allocations accordingly.



```
BB1
R0 = load[R4]
...
... = R0 ...
R1 = ...
R2 = ...
R3 = ...
... = R1 ...
... = R1 ...
... = R1 ...
R4 = R4 + STRIDE
ConditonalBr BB1
```

(c) Hybrid-amenable code. The RSP component can capture register accesses for sequential arithmetic instructions in the loop body. The RC component can capture accesses to the loop-carried value in R4.

Fig. 5. Examples of code for which an RSP, RC, and hybrid would be particularly effective

which register writes and reads will occur at runtime. This limits the compiler's ability to determine how much energy different scratchpad allocations will save and can lead to suboptimal allocation decisions. Profiling could help with this problem by providing the compiler with information about which instructions and register accesses are most likely to be encountered at runtime. However, we note that the RSP will be unable to adapt in the case of threads that follow less common control flow paths. Furthermore, the compiler must select allocations conservatively such that correctness is maintained no matter which control flow paths threads follow at runtime. For instance, the compiler cannot remap a register read to the RSP unless it can statically guarantee that a previously executed instruction wrote a valid value for this register to the RSP. This prevents the scratchpad from capturing register accesses that could be serviced if dynamic information were incorporated. In contrast, control flow is completely transparent to the cache, which just sees a series of register accesses and therefore only responds to the control flow paths that are taken at runtime.

3) Dynamically Shared Register Storage: The compiler is also not certain about when instructions will be executed dynamically. This stems from uncertainty about which control flow paths will be executed, how long it will take for dependencies on long latency-operations such as loads from global memory to resolve, and what decisions the warp scheduler will make at runtime. As mentioned previously, a strand boundary is inserted before dependencies on long-latency operations. This is done so that scratchpad space will not be idly occupied by a warp that is waiting for such a dependency to resolve. However, this is conservative, as it is possible that the long latency dependency will have resolved before execution reaches it. Scratchpad allocations are also conservative in that scratchpad capacity is equally divided between the maximum number of warps that could simultaneously issue instructions from the scheduler. However, some active warps may execute different segments of code at the same time in hardware or issue instructions more frequently than others. Therefore, some warps may have larger or smaller immediate-term register working sets than others.

An opportunity exists for the warps to dynamically share register capacity based on their immediate-term working sets, allowing more total register accesses to be captured across the executing warps. However, the compiler cannot take advantage of this because it cannot know which warps will execute which code at which time. In contrast, RC capacity is dynamically shared between all warps. If a warp stalls at runtime, the cache's LRU mechanism allows it to reclaim the capacity occupied by that warp's registers if the space is needed. Figure 4 compares the register energy consumed for a cache where each active warp is assigned a private, equally-sized cache segment as is done in an RSP and for a cache where capacity is dynamically shared between warps. These results show that a dynamic sharing of register storage is able to achieve greater energy savings than fixed partitioning. An RC can incorporate this beneficial feature, but an RSP must statically divide its capacity between the active warps.

D. Advantages of a Hybrid Design

The above discussion suggests that for kernels with features such as complex control flow and frequent long latency operations, like the example shown in Figure 5a, an RSP is likely to be ineffective at reducing register energy. We expect that an RC will outperform an RSP in such cases. For

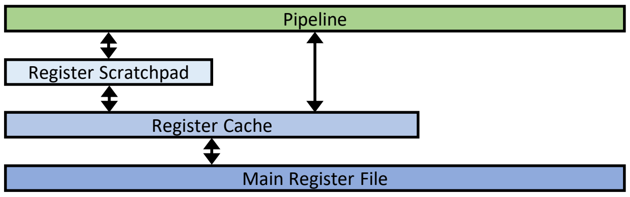


Fig. 6. Hybrid register hierarchy

kernels whose dynamic execution consists mostly of sequential arithmetic operations, such as the one shown in Figure 5b, an RSP will likely save more energy than an RC. This is because the compiler can accurately predict future register accesses and proactively allocate registers to the scratchpad in these situations. Because a hybrid design includes both an RC and an RSP, it will be at least somewhat effective in either scenario. This allows the hybrid design to capture register accesses across a broader range of code patterns than a cache-only or scratchpad-only design.

More significantly, the hybrid design benefits from a fine-grained synergy between the cache and the scratchpad. It is often the case that within a single kernel or segment of code some register accesses may be allocated to the scratchpad while others are more appropriately serviced by the cache. For instance, a strand may cover a portion of code that contains sequential arithmetic that the RSP captures more effectively than the RC would. At the same time, the RC may be able to capture register communication between strands or register accesses that the compiler failed to allocate to the RSP because of uncertainty about dynamic behavior.

An example of this is shown in Figure 5c. The loop body in this example contains a sequence of arithmetic instructions in which R1 is written and then read repeatedly. The compiler would proactively allocate R1 to the RSP and guarantee that it is not evicted until after these reads. R4 stores a loop counter that is read, incremented, and written at the end of the loop. The scratchpad is not able to communicate the value stored at the end of one loop iteration to the read of this value at the end of the next iteration because multiple strand boundaries are crossed in this span. However, strand boundaries do not affect the RC, which is therefore able to service these accesses to R4. Thus, a hybrid design could capture more register accesses than a cache-only or scratchpad-only design in this scenario. We note that in order for this synergy between the cache and the scratchpad to occur, each structure must capture register accesses that the other does not or cannot. This suggests that the hybrid design is most likely to outperform both the cache- and scratchpad-only designs for benchmarks where each single-structure design is at least moderately effective.

III. Design

A. Hardware

1) The Register Hierarchy: Our hybrid design adds both a register cache and register scratchpad to the baseline GPU. Some RSP allocations entail copying a register from the backing store to the RSP or writing back a dirty register from the RSP to the backing store. We enable the RC to intercept these accesses by organizing the register structures into a hierarchy in which the RSP is backed by the RC, which in turn is backed by the MRF. This is shown in Figure 6. Note that because the scratchpad uses a distinct segment of the name space, register access outside of this segment bypass the scratchpad. In Section IV-D, this design is compared to alternative designs in which the RC and RSP are organized in parallel, with both structures directly backed by the MRF. The hierarchical design achieves greater energy savings than the parallel approaches.

2) Register Cache and Scratchpad Policies: The designs of the RC and RSP components are based on those described in prior work [4], [5]. We implement a two-way set associative RC that uses an LRU replacement policy. The cache is write-allocated, meaning that the destination register of every executed instruction is allocated to the cache. Read-allocation was investigated, but resulted in cache pollution and reduced cache effectiveness. Because threads are grouped into warps whose instructions execute in lock-step in the hardware, both RC and RSP capacity is allocated at warp-granularity. To avoid unnecessarily writing dead register definitions back to the MRF, static-liveness hints are passed from the compiler to the hardware as proposed in prior work [4]. If this hint indicates that a register definition's life span ends after a final read, the cache invalidates an entry if it contains this register after reading the value.

RSP allocation decisions are dictated by the compiler and are described in detail in Section III-B. Here, we note that a number of changes may be made to the compiler's strategy to account for the presence of the cache in the hybrid design. For instance, part of the advantage of including both structures in the hybrid design is that the RSP can choose not to evict a register that will be accessed frequently some time in the future whereas the RC cannot account for future accesses. In cases where register accesses occur in immediate succession, such as a write followed directly by a read, the scratchpad may offer less of an advantage over the cache. One possible optimization would be to place a lower priority on allocating such accesses to the RSP in order to preserve capacity for accesses the RC is unlikely to capture. This technique and a number of similar approaches were investigated, but they had only a negligible impact on the energy savings achieved by the hybrid design. We therefore conclude that the hybrid design is able to effectively reduce register energy without altering the scratchpad allocation algorithm.

3) Integration into a Streaming Multiprocessor: Every SM in the GTX 980 contains 4 schedulers. Each warp is mapped to one scheduler for its entire lifetime and only issues instructions from this scheduler. Therefore, a warp's registers will only be accessed by instructions issued from a single scheduler. As such, private register cache and scratchpad storage is provisioned for each scheduler. A scheduler may issue up to two instructions in a single cycle and each instruction may write 1 register and read 3 registers. To avoid limiting performance, 2 write ports and 6 read ports are provisioned for the temporary register storage. This high degree of multi-porting results in higher access energy than a design with a lower port count. However, the total energy of the register

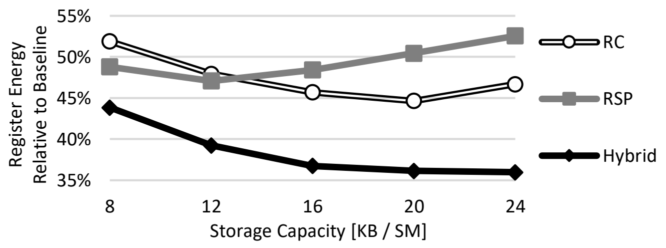


Fig. 7. Geometric mean of register energy relative to the baseline across all Rodinia benchmarks for cache-only, scratchpad-only, and hybrid designs with 8, 12, 16, 20, and 24 KB / SM of storage. Capacity is evenly divided between the RC and RSP subcomponents of the hybrid design.

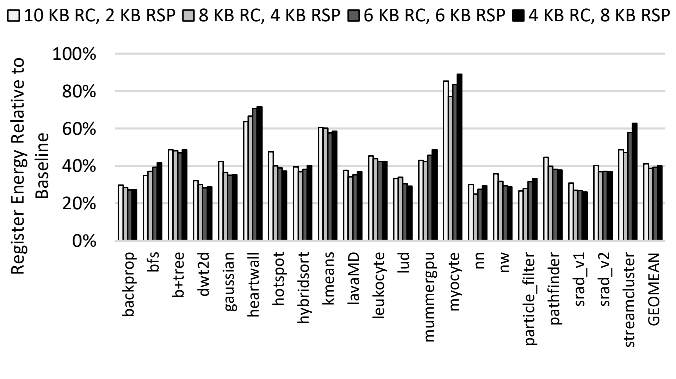


Fig. 8. Register energy for various hybrid register cache / scratchpad configurations with total storage of 12 KB / SM

hierarchy is dominated by the MRF, so this is not a major detriment.

4) Scaling the Hybrid Design: Our thesis is that, for a given amount of register storage, a hybrid register cache / scratchpad will more effectively reduce register energy than a cache-only or scratchpad-only design of the same capacity. Therefore, we will evaluate the three designs with the same amount of storage allocated for each. Figure 7 shows the register energy of each approach relative to the baseline for a range of storage capacities. In this survey, the capacity allocated to the hybrid design is equally divided between its RC and RSP subcomponents. The results show that increasing RSP capacity above 12 KB per streaming multiprocessor results in higher energy consumption. This is because the energy saved by eliding additional MRF accesses is offset by the increased energy required to access the RSP beyond this point. We wish to compare the hybrid design against the best possible implementation of this technique, so we will evaluate the three approaches with 12 KB / SM of capacity. However, we note that the hybrid design consistently achieves higher energy savings than the cache-only and scratchpad-only designs for all capacities examined.

It is not intuitively obvious how storage capacity should be divided between the RC and RSP subcomponents of the hybrid design. As discussed above, we provision 12 KB / SM of total capacity for the hybrid design. This translates to 6 warp-wide registers per active warp. Here, we evaluate allotting 2 KB, 4 KB, 6 KB, and 8 KB of this storage to the RSP with the remaining storage alloted to the RC. 2 KB / SM of storage provides one warp-wide register for each active warp[1], so smaller increments of storage division do not represent compelling points in the design space. The results are shown in Figure 8. The greatest energy savings are achieved with 8 KB alloted to the RC and 4 KB to the RSP. This translates to 4 registers per active warp of cache storage and 2 registers per active warp of scratchpad storage. However, we note that the optimal division of capacity varies by benchmark. This is because some benchmarks exhibit more cache-amenable behavior while others exhibit more scratchpad-amenable behavior. Using Cacti [12], we estimate that this hybrid configuration increases GPU die area by 4.7%, compared to 5.7% and 3.6% for the RC and RSP designs, respectively.

B. Compiler Support

The compiler is responsible for remapping registers to the scratchpad with the goal of minimizing the energy consumed by register writes and reads. We implement a compiler algorithm to do this that is modeled after the one described by Gebhart et al. [5]. The compiler's task consists of several subtasks: strand formation, allocation candidate identification, and allocation candidate selection.

As discussed in Section II-C, the compiler divides the kernel code into segments called strands and will only remap a register to the scratchpad within the span of a single strand. The compiler forms strands by iterating over the program in SIMT execution order, adding each instruction to the current strand as it proceeds. The current strand ends when one of the following is encountered:

- A basic block with any control flow input edges that are not contained in the strand. This ensures that each strand has only a single entry point.
- A basic block that could be executed multiple times in SIMT execution order if included in the strand
- A backwards branch or the target of a backwards branch
- A function call or a return, unless it is a predicated return that exits the kernel
- An instruction that has a dependency on a long latency operation if it is possible that this will be the first dynamically executed instruction with this dependency

The strand formation process repeats until the set of strands covers the entire kernel.

Once a strand has been formed, the compiler identifies every possible maximal register allocation within the strand. We say an allocation is maximal if extending its range would necessarily result in an interval in which the allocation occupies scratchpad capacity to hold a value that will not be read from the scratchpad. There are two primary types of scratchpad allocations: Write allocations and read allocations. The range of a write allocation begins with a strand instruction writing its result to the scratchpad and ends with a read of the register from the scratchpad. If the lifetime of a register definition written to the scratchpad extends beyond this final read, the register value must be written to the backing store.

[1] 128 bytes / warp-wide register, 4 active warps / scheduler, 4 schedulers / SM

We refer to these cases as live-out allocations. A write-allocated register may be written multiple times within the range in our implementation, whereas it may only be written once at the start of the range in the implementation described by Gebhart et al. [5]. The range of a read allocation begins with a register read that transfers the value from the backing store to both the functional unit pipeline and the scratchpad. The range ends with a later read. Read-allocated registers may not be written in the RSP and thus never need to be transfered from the RSP to the backing store. Registers may only be allocated to the RSP within a strand and registers may not be communicated through the RSP across strand boundaries. For registers that are allocated to the RSP but whose lifetimes span strand boundaries, the compiler is responsible for directing register transfers between the RSP and the backing store by selecting live-out and read allocations as appropriate. In these cases, the compiler must be able to statically guarantee that in all possible runtime scenarios register definitions are transfered in a way that ensures correctness.

Finally, the compiler selects which allocation candidates to map to the scratchpad. The goal of this selection process is to maximize energy savings with the limitation that the available capacity of the scratchpad cannot be exceeded. For each allocation candidate, the compiler first calculates the register access energy that would be saved by mapping the allocation to the scratchpad. A score is assigned to each candidate by dividing its energy savings by the length of its range and the candidates are placed in a list sorted by this score. The compiler then iterates through the sorted list, greedily mapping each allocation to the scratchpad if there is space available for the allocation's entire range. If space is not available for an allocation's entire range, the range may be reduced and the allocation reinserted into the sorted list. The range of a write allocation is first reduced by iteratively removing read and write accesses from the tail of the range. If the entire range is eliminated without mapping the allocation to the scratchpad, the accesses from the original range are restored except for those before the second write and the process starts over. For a read allocation containing N register reads, the range is reduced by producing new allocations with ranges corresponding to the subsets of the original range with N-1 consecutive reads. If none of these new allocations are selected, new allocations with N-2 reads are created and so on. When an allocation with a reduced range is mapped to the RSP, the portions of the original range that were removed are used to generate new allocation candidates. This is more sophisticated than the technique describe in prior work [5], which only removes accesses from the tail for both write and read allocations and does not generate new allocations to cover the excluded accesses.

IV. Evaluation

A. Methodology

We simulate our design using GPGPU-sim 3.2.2 [14] configured to model an Nvidia GTX 980 [6]. The simulation parameters are listed in Table I. A custom compiler framework is used to analyze the code and map registers to the scratchpad as described in Section III-B. We model the energy consumed by the RC and RSP components using Cacti 6.5 [12] at 28 nm. For consistency, the energy of the MRF is also modeled with Cacti. The energy consumption of the other GPU components is estimated using GPUWattch [3]. Our design is evaluated across the applications in the Rodinia benchmark suite [11].

TABLE I
GPGPU-SIM SIMULATION PARAMETERS

Architecture	Nvidia Maxwell
Number of SMs	16
Warps	64 warps/SM, 32 threads/warp
Warp schedulers	4 schedulers/SM, issue max. 2 instructions/scheduler/cycle, Two-level scheduler with 4 active warps/scheduler and round robin outer-level policy plus GTO inner-level policy for RC, RSP, and hybrid designs, GTO scheduler for the baseline
Register file	256 KB/SM, 16 banks, 1 read-write port/bank
RFC and RFSP	12 KB total/SM, 128 B/line, 2 write ports, 6 read ports, 2-way associate with LRU replacement (cache only)
Shared memory	96 KB/SM
L1 cache	48KB, 32 MSHRs, 1 request/cycle, global data accesses bypassed [13]
L2 cache	2048 KB, 4 memory partitions, 224 GB/s B/W

We compare the effectiveness of our hybrid design against an RC and an RSP that are based on the designs presented in prior work [4], [5]. 12 KB / SM of temporary register storage is provisioned as discussed in Section III-A4, with 8 KB / SM alloted to the RC and 4 KB / SM alloted to the RSP subcomponents of the hybrid design.

B. Energy Savings and MRF Access Elision

The motivating idea behind our hybrid design is that RCs and RSPs employ complimentary allocation schemes and, as a result, a hybrid approach will reduce register energy more effectively than either alone. We further expand this hypothesis and make three claims:

- Claim 1: In the case of benchmarks for which both a cache-only and scratchpad-only design are similarly effective, the complimentary nature of the RC and the RSP usually produces a synergistic effect that benefits the hybrid design.
- Claim 2: Due to this synergy, fewer register accesses must be serviced by the MRF with the hybrid design compared to a cache-only or scratchpad-only design.
- Claim 3: As a result, the hybrid design more effectively reduces register energy.

We now investigate each of these claims.

1) Energy Savings: In Figure 9, we show the energy consumed by the entire register hierarchy (MRF, RC, and RSP) relative to the baseline design, which employs a standard greedy-then-oldest (GTO) scheduler rather than a two-level scheduler. Across all benchmarks, the geometric mean of the register energy relative to the baseline is 47.9% for the RC, 47.1% for the RSP, and 38.7% for the hybrid design. Further, the geometric mean of the total GPU energy relative to the baseline is 94.3% for the RC, 93.9% for the RSP, and 92.8% for the hybrid design.

Accesses to the RC and RSP in the hybrid design consume less energy than accesses in the cache-only and scratchpad-

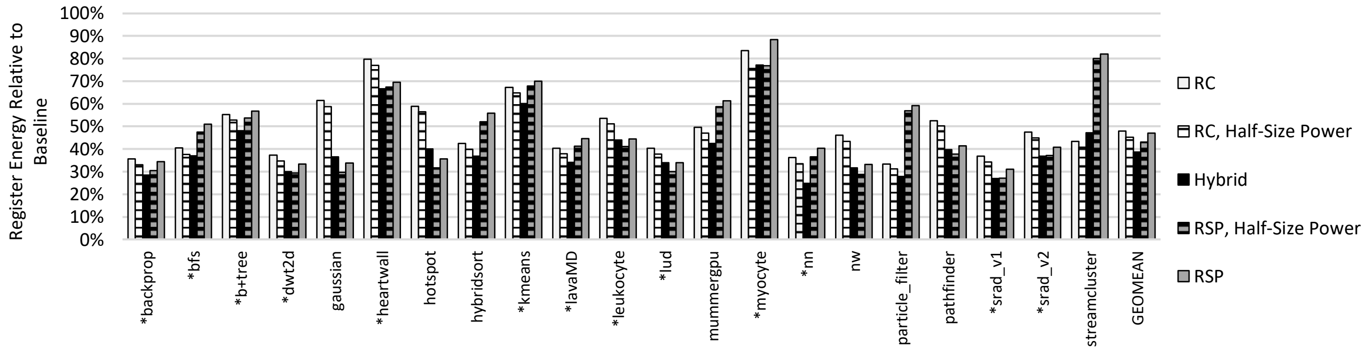


Fig. 9. Register energy for cache-only [4], scratchpad-only [5], and hybrid designs. * = balanced benchmark

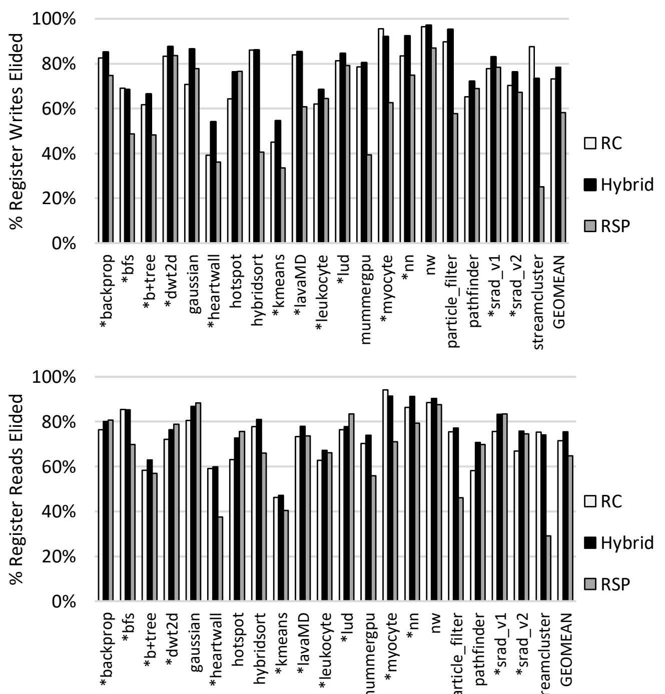


Fig. 10. Register write and register read elision rates for cache-only, scratchpad-only, and hybrid designs. * = balanced benchmark

only designs with the same total capacity because the individual structures in the hybrid design are smaller. While this is beneficial, it is not the primary cause of the hybrid design's superior energy savings. Furthermore, it is conceivable that the RC or the RSP in the single-structure designs could be modified with partitioning or power gating schemes such that their energy consumption would be similar to that of one of the smaller structures in the hybrid design. In order to verify that our hybrid design offers an advantage even in this case, we again simulate the RC and RSP approaches with 12 KB / SM of storage, but model their energy consumption as if only 6 KB / SM had been provisioned. These results are shown with the label "half-size power" in Figure 9. Under these conditions, the geometric mean of the register energy relative to the baseline is 45.1% for the RC and 43.1% for the RSP. Thus, our hybrid design saves more register energy than the single-structure designs even after controlling for the difference in structure sizes, verifying Claim 3.

2) MRF Access Elision: The hybrid design is more effective at reducing register energy than the cache-only or scratchpad-only designs primarily because it is better able to elide accesses to the MRF. This is illustrated in Figure 10, which shows the percent of register writes and reads that are elided by each scheme. Across all benchmarks, the geometric mean of the percent of register writes elided is 73.3% for the RC, 58.1% for the RSP, and 78.3% for the hybrid design. The geometric mean of the percent of register reads elided is 71.5% for the RC, 64.8% for the RSP, and 75.5% for the hybrid design. These results support Claim 2.

We note that the geometric mean of the percent of register accesses elided by the scratchpad is lower than for the cache. However, scratchpad accesses do not require tag stores or comparisons, so the RSP still reduces register energy more effectively than the RC. Further, the scratchpad elides substantially more register accesses than the cache for some benchmarks, such as hotspot.

3) RC and RSP Synergy in the Hybrid Design: The above results demonstrate that the hybrid design reduces the number of MRF accesses, leading to lower register energy consumption. We now focus on the mechanisms that enable it to do so. The motivating idea behind our hybrid design is that RCs and RSPs employ complimentary allocation schemes. When the two are combined, a synergistic effect can result that allows the hybrid design to capture more register accesses for the same segment of code than a cache or scratchpad alone. This synergy is particularly likely to occur when both an RC and RSP are at least moderately effective and therefore both able to contribute to the set of register accesses captured.

For this evaluation, we will say that a benchmark is balanced if the difference between the register energy consumption of the cache- and scratchpad-only designs is less than 10% of the register energy of the baseline. For the half-size power models, 13 of the benchmarks are balanced. These are marked with a * in Figures 9 and 10. We observe that of these 13 benchmarks, the hybrid design consumes less energy than either half-size power single-structure design for 9 benchmarks, elides more register writes for 11 benchmarks, and elides more register reads for 7 benchmarks. Thus, the hybrid design outperforms both single-structure designs for the majority of balanced

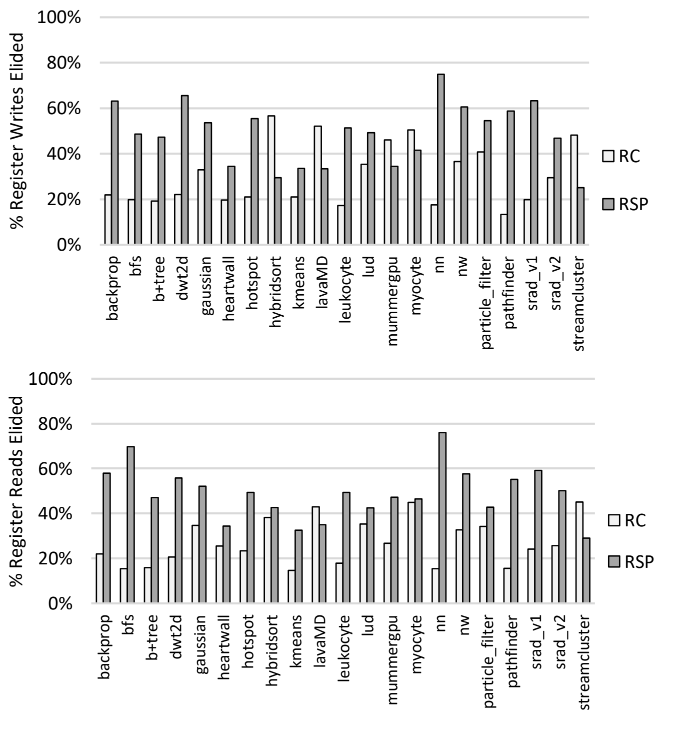


Fig. 11. Percent of kernel register writes and register reads elided by the RC and RSP subcomponents of the hybrid design. The RSP generally elides more accesses than the RC.

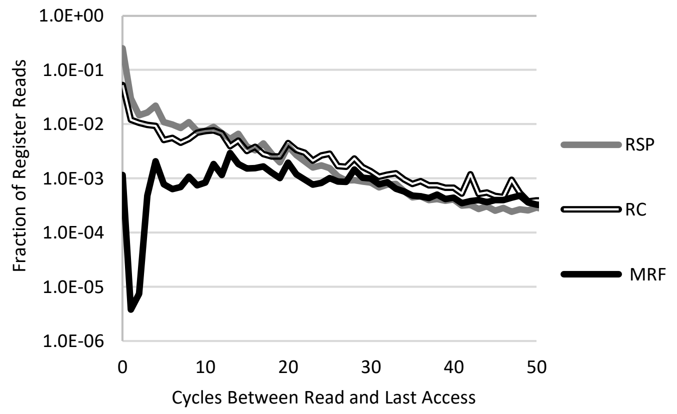


Fig. 12. Fraction of kernel register reads captured by structures in the hybrid design versus cycles between register read and last register access averaged across all Rodinia benchmarks (log scale). The RSP component captures most high-locality accesses, leaving lower locality accesses for the RC to capture.

benchmarks, verifying Claim 1.

The hybrid design is less likely to outperform both the cache- and scratchpad-only designs for unbalanced benchmarks where one of these single-structure approaches is substantially more effective than the other. For example, the majority of the execution time for streamcluster is spent in a tight loop with a long-latency dependency in the middle, little register reuse within the loop body, and a large number of loop-carry registers that span the backwards branch. As a result, the cache outperforms the scratchpad for this benchmark. Conversely, hotspot spends the majority of its execution time in a large loop with simple control flow and large amounts of sequential arithmetic operations. As a result, the scratchpad saves much more energy than the cache in this case. For both of these benchmarks, the hybrid design achieves savings that are between those of the cache and the scratchpad. This is because one of the structures in the hybrid design is ineffective and the other has less capacity than its single-structure counterpart. However, for 3 of the 8 unbalanced benchmarks, the hybrid design still manages to achieve greater energy savings than either single-structure design.

C. Analysis of Cache and Scratchpad Effectiveness Within the Hybrid Design

The above discussion analyzes the overall effectiveness of the hybrid register cache / scratchpad. We now turn our attention to the effectiveness of the individual RC and RSP subcomponents of the hybrid . Figure 11 shows the percent of all register writes and reads elided by the cache and the scratchpad within the hybrid design. We define the percent of register writes elided by a particular structure as the number of writes to the structure from a higher level of the hierarchy shown in Figure 6 minus the number of writebacks from this structure to a lower level of the hierarchy divided by the total number of register writes for the kernel. We define the percent of register reads elided by a particular structure as the number of reads serviced by the structure divided by the total number of register reads for the kernel. These results show that the scratchpad tends to elide more register accesses than the cache in our hybrid design despite the fact that the best hybrid configuration found allots more storage space to the cache than to the scratchpad.

The reason for this is that many easily captured register accesses are serviced by the scratchpad. This is illustrated in Figure 12, which shows the distribution of register reads serviced by each structure in the register hierarchy versus the number of cycles between each read and the most recent prior access to the same register. Within 20 cycles of the last access, the RSP generally captures more reads than the RC. We further note that the share of reads serviced by the RSP declines more rapidly as the number of cycles increases than does the share captured by the RC. Because the scratchpad captures most of the high-locality accesses, the register accesses remaining for the cache to intercept exhibit lower locality. This results in reduced RC effectiveness in the hybrid design. For instance, the geometric mean hit rate for register reads in the cache component of the hybrid design is 51.3% compared to 71.5% for the cache-only design. However, we note that the cache component is still critical to the high overall effectiveness of the hybrid design because it captures register accesses that the scratchpad component does not, such as those that exhibit lower locality.

D. Comparison of Parallel and Hierarchical Designs

The RC and RSP components of the hybrid design are organized in a hierarchy as shown in Figure 6. An alternative approach is to implement the RC and RSP in parallel, with both structures directly backed by the MRF. However, some scratchpad allocations (live-out and read allocations, discussed in Section III-B) entail copying a register from the backing store to the RSP or writing back a dirty register from the RSP to the backing store. If the RC and RSP are implemented in parallel, care must be taken to avoid the case where a register is written to the RC or RSP but a stale definition of the register is later read from the other structure. One way to avoid this is to evict a register from the RC before allocating the same register to the RSP unless both copies will be clean. However, this increases MRF accesses and reduces register energy to only 43.7% of the baseline compared to 38.7% for the hierarchical design. Another parallel solution is to disable live-out and read allocations, allocating registers to the RSP only if they will reside there for their entire lifetime. This limits which register accesses can be serviced by the RSP, resulting in register energy that is 39.8% that of the baseline. This is not quite as effective as the hierarchical design, which can service accesses to a register from different structures at different points in the register's lifetime.

Although internally the MRF, RC, and RSP form a hierarchy, the pipeline can access the RC and RSP in parallel. Further, the transfer of registers between the RSP and lower levels of the hierarchy happens preemptively in the background and the compiler guarantees that the pipeline will never "miss" when it accesses a register in the RSP. Therefore, pipeline register access latency is only increased compared to the baseline when an access misses in the RC and then must be serviced by the MRF. Using Cacti [12], we estimate that MRF accesses take 0.82 ns, RC accesses take 0.22 ns, and RSP accesses take 0.17 ns. Thus, the worst case access latency is slightly over 1 ns. The clock frequency of the GTX 980 is 11126 MHz [6], giving a clock period of 0.89 ns. Therefore, an extra cycle is required for the uncommon case that an access must be serviced by the MRF. Prior work by Gebhart et. al. [4] addressed this issue by adding a pipeline stage, which was shown to have a negligible impact on performance in a throughput-oriented GPU architecture.

V. Related Work

Gebhart et al. [4], [5] proposed the register cache and register scratchpad for GPUs. Our hybrid approach combines the advantages of the RC and RSP to achieve higher energy savings. Sadrosadati et. al. [8] prefetch registers into a register cache when a warp becomes active, but this technique targets register access latency, not energy.

Other works propose altering the MRF architecture to save energy. Kloosterman et. al. [9] replace the MRF with a smaller, memory-backed operand staging unit that limits spills and fills using intelligent scheduling. Jeon et. al. [7] reduce the MRF size by enabling physical registers to be used by different warps over time. Abdel-Majeed et. al. [15] propose placing registers in a low-power state between accesses and disabling register accesses by inactive warp lanes. Our approach saves energy without altering the standard register file design.

Some works propose reducing register energy by implementing the MRF with novel memory technologies [16]–[21]. These technologies present challenges such as long access latencies and periodic refreshes. Our design's small size allows it to save energy using traditional memory technology, avoiding these challenges.

Warp threads often store identical or similar values to a register, which may be compressed. Register energy can be reduced by storing compressed registers in small, cache-like structures [22], [23] or by activating fewer MRF banks or sub-arrays to access the compressed registers [24], [25]. Our approach can reduce register energy regardless of register values.

VI. Conclusion

The large size of GPU register files results in high-energy register writes and reads. This presents a challenge to maintaining energy-efficiency in GPUs as the size of the register file continues to grow over successive generations. Prior work has proposed augmenting GPUs with register caches and scratchpads. While both of these techniques reduce the amount of energy needed to accesses registers, we observe that each approach is more effective for some types of register accesses than for others. We propose a hybrid design that leverages the dynamic allocation capabilities of the cache and the proactive allocation capabilities of the scratchpad. This allows the hybrid register cache / scratchpad to capture a broader range of register accesses than either single-structure approach. As a result, our technique is able to reduce register accesses energy to 38.7% of the baseline, compared to 47.9% for a cache-only design and 47.1% for a scratchpad-only design.

References

[1] J. Bailey, J. Kloosterman, and S. Mahlke, "Scratch that (but cache this): A hybrid register cache/scratchpad for gpus," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2779–2789, Nov 2018. [Online]. Available: https://ieeexplore.ieee.org/document/8413125

[2] Nvidia, "Nvidia tesla v100 gpu architecture," White Paper, Aug 2017. [Online]. Available: http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[3] 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. New York, NY, USA: ACM, 2013, pp. 487–498. [Online]. Available: http://doi.acm.org/10.1145/2485922.2485964

[4] M. Gebhart, D. R. Daniel, 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. New York, NY, USA: ACM, 2011, pp. 235–246. [Online]. Available: http://doi.acm.org/10.1145/2000064.2000093

[5] 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*, ser. MICRO-44. New York, NY, USA: ACM, 2011, pp. 465–476. [Online]. Available: http://doi.acm.org/10.1145/2155620.2155675

[6] Nvidia, "Whitepaper: Nvidia geforce gtx 980," White Paper, 2014. [Online]. Available: http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_980_Whitepaper_FINAL.PDF

[7] H. Jeon, G. S. Ravi, N. S. Kim, and M. Annavaram, "Gpu register file virtualization," in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO-48. New York, NY, USA: ACM, 2015, pp. 420–432. [Online]. Available: http://doi.acm.org/10.1145/2830772.2830784

[8] M. Sadrosadati, A. Mirhosseini, S. B. Ehsani, H. Sarbazi-Azad, M. Drumond, B. Falsaf, R. Ausavarungnirun, and O. Mutlu, "Ltrf: Enabling high-capacity register files for gpus via hardware/software cooperative register prefetching," in *Proceedings of The 23rd ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS'18. ACM, 2018. [Online]. Available: http://users.ece.cmu.edu/~rausavar/pubs/ltrf-asplos18.pdf

[9] J. Kloosterman, J. Beaumont, D. A. Jamshidi, J. Bailey, T. Mudge, and S. Mahlke, "Regless: Just-in-time operand staging for gpus," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17. New York, NY, USA: ACM, 2017, pp. 151–164. [Online]. Available: http://doi.acm.org/10.1145/3123939.3123974

[10] 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)*, ser. MICRO-49, Oct 2016, pp. 1–14.

[11] 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 *2009 IEEE International Symposium on Workload Characterization (IISWC)*, Oct 2009, pp. 44–54.

[12] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," White Paper, April 2009. [Online]. Available: http://www.hpl.hp.com/techreports/2009/HPL-2009-85.pdf

[13] Nvidia. Cuda c programming guide. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

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

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

[16] 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 *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY, USA: ACM, 2013, pp. 344–355. [Online]. Available: http://doi.acm.org/10.1145/2485922.2485952

[17] 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*, ser. ISLPED '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 3–8. [Online]. Available: http://dl.acm.org/citation.cfm?id=2648668.2648673

[18] 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 *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ser. ISCA '11. New York, NY, USA: ACM, 2011, pp. 247–258. [Online]. Available: http://doi.acm.org/10.1145/2000064.2000094

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

[20] 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 *Proceedings of the 51st Annual Design Automation Conference*, ser. DAC '14. New York, NY, USA: ACM, 2014, pp. 196:1–196:6. [Online]. Available: http://doi.acm.org/10.1145/2593069.2593137

[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)*, Feb 2017, pp. 589–600.

[22] 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. New York, NY, USA: ACM, 2013, pp. 130–141. [Online]. Available: http://doi.acm.org/10.1145/2485922.2485934

[23] P. Xiang, Y. Yang, M. Mantor, N. Rubin, L. R. Hsu, and H. Zhou, "Exploiting uniform vector instructions for gpgpu performance, energy efficiency, and opportunistic reliability enhancement," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ser. ICS '13. New York, NY, USA: ACM, 2013, pp. 433–442. [Online]. Available: http://doi.acm.org/10.1145/2464996.2465022

[24] 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. New York, NY, USA: ACM, 2015, pp. 502–514. [Online]. Available: http://doi.acm.org/10.1145/2749469.2750417

[25] 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)*, Feb 2017, pp. 601–612.

Jonathan Bailey received a B.S. degree in computer engineering from Clemson University, Clemson, SC, USA in 2015 and an M.S. in computer science and engineering for the University of Michigan, Ann Arbor, MI, USA in 2016. He is currently pursuing a Ph.D. in computer science and engineering at the University of Michigan.

He was an Information Technology Intern at The Boeing Company in 2014 and an Intern with the Media Processing Group at Arm in 2017. His research interests include computer architecture, compilers, GPUs, and hardware/software co-design.

Mr. Bailey received the Rhodes Most Outstanding Junior and Senior in Computer Engineering Awards and the Faculty Scholarship Award from Clemson University.

John Kloosterman is a Lecturer at the University of Michigan, Ann Arbor, MI, USA. He received the Ph.D. degree in computer science and engineering from the University of Michigan in 2018. His research interests include computer architecture and computer science education.

Scott Mahlke received the Ph.D. degree in electrical engineering from the University of Illinois, Champaign, IL, USA in 1997.

He is currently a Professor in the Electrical Engineering and Computer Science Department, University of Michigan, Ann Arbor, MI. He leads the Compilers Creating Custom Processors Research (CCCP) Group, http://cccp.eecs.umich.edu, focusing on the areas of energy-efficient processor design, hardware accelerators, and reliable system design.

Dr. Mahlke was awarded the Young Alumni Achievement Award from the University of Illinois in 2006, the Most Influential Paper Award from the International Symposium on Computer Architecture in 2007, and the Test of Time Award from the International Symposium on Microarchitecture in 2014.