2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture

# RegMutex: Inter-Warp GPU Register Time-Sharing

Farzad Khorasani* Hodjat Asghari Esfeden† Amin Farmahini-Farahani‡ Nuwan Jayasena‡ Vivek Sarkar*

*Georgia Institute of Technology, School of Computer Science, Atlanta, GA, USA
{farkhor, vsarkar}@gatech.edu
†University of California Riverside, Department of Computer Science, Riverside, CA, USA
hasgh001@ucr.edu
‡AMD Research, Santa Clara, CA, USA
{afarmahi, nuwan.jayasena}@amd.com

***Abstract*—Registers are the fastest and simultaneously the most expensive kind of memory available to GPU threads. Due to existence of a great number of concurrently executing threads, and the high cost of context switching mechanisms, contemporary GPUs are equipped with large register files. However, to avoid over-complicating the hardware, registers are statically assigned and exclusively dedicated to threads for the entire duration of the thread's lifetime. This decomposition takes into account the maximum number of live registers at any given point in the GPU binary although the points at which all the requested registers are used may constitute only a small fraction of the whole program. Therefore, a considerable portion of the register file remains under-utilized.**

**In this paper, we propose a software-hardware co-mechanism named RegMutex (Register Mutual Exclusion) to share a subset of physical registers between warps during the GPU kernel execution. With RegMutex, the compiler divides the architected register set into a base register set and an extended register set. While physical registers corresponding to the base register set are statically and exclusively assigned to the warp, the hardware time-shares the remaining physical registers across warps to provision their extended register set. Therefore, the GPU programs can sustain approximately the same performance with the lower number of registers hence yielding higher performance per dollar. For programs that require a large number of registers for execution, RegMutex will enable a higher number of concurrent warps to be resident in the hardware via sharing their register allocations with each other, leading to a higher device occupancy. Since some aspects of register sharing orchestration are being offloaded to the compiler, RegMutex introduces lower hardware complexity compared to existing approaches. Our experiments show that RegMutex improves the register utilization and reduces the number of execution cycles by up to $23\%$ for kernels demanding a high number of registers.**

***Keywords*-compiler; GPGPU; GPU; microarchitecture; register; register file; RegMutex; time-multiplexing; time-sharing; warp;**

## I. Introduction

Registers are the fastest available memory to the threads in a machine executing a program. Register are being kept in-core closely coupled with the ALUs and usually are the most expensive form of memory (per-byte) in a machine. The set of registers for a processor are packed in a structure called *register file*. In GPUs, in order to enable concurrent residence of thousands of threads for massive Thread-Level Parallelism (TLP), the architecture employs a very large SRAM storage structure as the register file. A considerable fraction of die area and chip power has to be dedicated to this structure [1], [2].

Nonetheless the allocation of physical registers to architected registers in the kernel binary is static, i.e., the maximum number of live registers at any given point determines the kernel's physical register demand, and is exclusive, i.e., a warp's physical registers are solely its own for the lifetime of the thread-block containing the warp. This allocation scheme carves a portion of the physical registers for the warp regardless of the fluctuations in the register usage by the warp. In other words, even if all the requested registers are live only for a few instructions, the hardware reserves all the allocated physical registers for the warp, thereby making them inaccessible by other warps. This results in underutilization of a large portion of the register file during GPU kernel execution and hence ignoring the potential performance gain opportunity.

A number of solutions have been proposed to remedy this issue. Most notably, Jeon *et al.* [3] built on the Register Renaming Table (RRT) idea from the CPU realm to proactively map architected registers to physical registers on-demand. However, this solution, as well as other work that suggest fundamental modifications to the structure of the register file and its allocation mechanism [4], [5], [6], impose significant hardware overheads. In addition, proposals such as the work of Jatala *et al.* [7] fail to address fluctuations in warp register demand during kernel execution, hence lack general applicability.

In this work, we present RegMutex (**Reg**ister **Mut**ual **Ex**clusion), a synergistic compiler-microarchitecture design that enables efficient register time-multiplexing between warps. In RegMutex, a subset of the architected registers are allocated on-demand as-a-whole and deallocated upon no demand. RegMutex utilizes the information gathered by compiler analysis to instruct the hardware for physical register allocation and deallocation. At compile time, RegMutex separates the group of kernel architected registers into a

2575-713X/18/$31.00 ©2018 IEEE
DOI 10.1109/ISCA.2018.00073

base register set and an extended register set. Using live-register analysis, the compiler determines the locations within the kernel where the number of live registers exceeds the size of the base register set and marks them as *acquire* points. Similarly, program points where the number of live registers falls equal to or below the size of the base register set are marked as *release* points. On the hardware side, the physical registers are allocated for the base register set whenever the warp is resident. The extended register set, on the other hand, is allocated physical registers only when the warp reaches an acquire point, and deallocated once the warp faces a release point. The extended register sets of all warps are allocated out of a communal pool of registers that is shared by all hardware-resident warps (the *shared register pool*).

RegMutex diminishes the pressure on the register file by eliminating the necessity of register file size accommodation with the maximum number of live registers at any point in the kernel. The warps proceed in the program as usual and will be blocked only when a large number of them have acquired the extended register set. A blocked warp will resume execution by acquiring the extended set as soon as one of other warps releases the shared resource. Essentially, the benefit of RegMutex can be viewed from two perspectives. First, RegMutex allows GPU programs to sustain approximately the same performance with a smaller hardware register file. Second, for programs that incur low SM occupancy due to excessive register usage, our technique enables higher number of concurrent warps to be resident in the hardware via sharing their register allocations with each other, leading to a superior performance. In other words, if a warp asks for a large number of architected registers, it can now co-reside with more warps on the SM. This paper makes the following contributions:

- We present RegMutex, a coordinated compiler-microarchitecture technique as a remedy for GPU register file underutilization due to static and exclusive physical register allocation.
- We describe the RegMutex compiler and microarchitecture support schemes and show that this synergistic design introduces much lower (less than $2\%$) hardware storage overhead compared to existing solutions.
- We analyze the effectiveness of our solution by implementing it in the GPGPU-Sim simulation framework. We show that RegMutex enhances the performance of kernels for which the occupancy is limited by high register demand, and makes the application performance resilient on architectures that supply small register files.

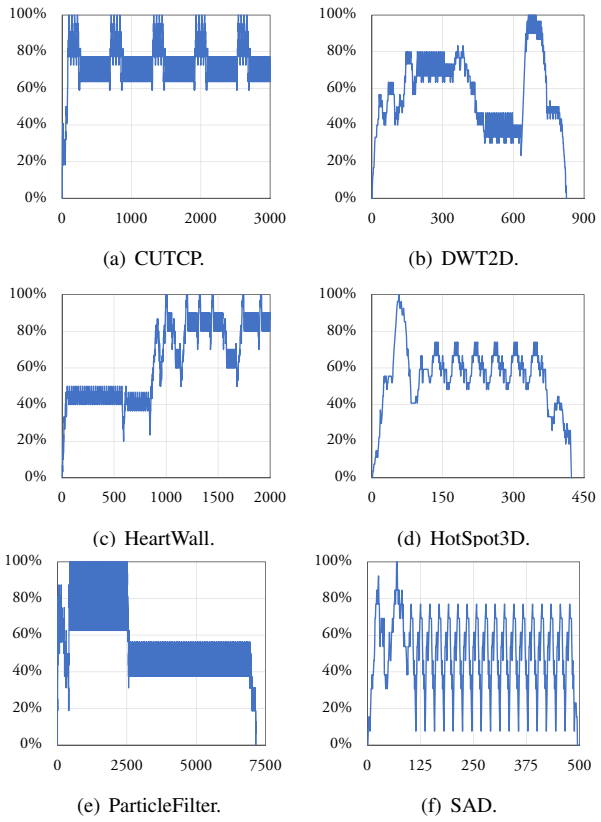
The rest of this paper is organized as follows. Section II expresses the motivation for a GPU register sharing approach. Section III describes our solution, elaborating RegMutex's compiler and micro-architectural design. Section IV presents the experimental evaluations, and Sections V and VI summarize related work and our conclusions.

## II. Motivation

A program, in its closest-to-machine language form, works with a set of registers. This set of registers are referred to as *architected registers*, and will be mapped to physical registers by the processor's hardware. To map architected registers to physical registers, CPUs utilize a mechanism called register renaming and a table called Register Rename Table (RRT). GPUs, on the other hand, use a simpler method for this purpose [8]. The mapping allows a simple $Y = X+B$ equation for each SIMD group (warp) to calculate its physical register indices where $B$ is the base address of the block of registers assigned at run time to the specific warp, $X$ is the architected register index (i.e., the offset into the block of registers), and $Y$ gives the physical register index. This simple mapping avoids the overhead of performing register renaming for thousands of concurrently running threads. The set of physical registers is statically reserved for the warp's life-time (i.e., $B$ is constant for the duration of the warp's execution), and becomes available for other threads only after the CTA (Cooperative Thread Array) to which the warp belongs retires [9].

One important drawback of the above scheme, especially compared to RRT, is physical register underutilization during kernel execution. The static reservation is conservative in a sense that it requests for the *maximum* number of registers that are alive at any point in the GPU program. However, during a GPU program execution by the warp, not all the reserved physical registers are alive at all times. In fact, the time interval in which all the requested physical registers are utilized may only be a small fraction of the kernel execution time. This is particularly true for GPU applications containing nested loops in which register consumption increases within inner loops. Figure 1 illustrates this claim by showing the percentage of live registers with respect to the allocated registers during the program execution for a sample thread and six GPU kernels. Here we define a register *live* if its value is used in later instructions. It is evident from the plots that for the majority of the program execution only subsets of the requested registers are alive, and therefore, a large portion of the thread's allocated registers remain unutilized. Figure 1 also shows that register utilization may fluctuate constantly due to the GPU code shape.

Another drawback of the aforementioned scheme is limiting the occupancy for GPU programs (kernels) with threads that require a high number of architected registers. Occupancy is described as the ratio of the number of warps residing on the Streaming Multiprocessor (SM) over the maximum number of warps that warp schedulers in the SM allow for residency. For example, on Nvidia Volta GPUs, there can be up to 64 warps residing on an SM [13]. The higher the occupancy, the larger the number of candidate warps to be executed by the SM at any given time. This enables the GPU cores to cover memory access latency




(a) CUTCP. (b) DWT2D. (c) HeartWall. (d) HotSpot3D. (e) ParticleFilter. (f) SAD.

Figure 1. The utilization of a sample thread's allocated register set during kernel execution. X axis shows the number of instructions executed by the thread and Y axis shows the percentage of live registers with respect to allocated registers. Results are extracted using our extension to *GPGPU-Sim* [10]. Applications are from Rodinia [11] and Parboil[12].

more effectively through having more concurrent warps (note that higher occupancy does not necessarily lead to better performance [14] due to possible side-effects such as cache pollution, but a low occupancy can cause resource underutilization). A warp that requires a high number of registers lowers the SIMD occupancy. It essentially disallows co-residency of other warps due to register file resource limitations, however the warp may need excessive registers only for a short period of the GPU program. In summary, these two drawbacks are two different faces of the same coin: registers are statically and exclusively reserved but not all of them are utilized at the same time.

Jatala *et al.* [7] propose a register sharing technique in which a few warps share the set of registers having higher-than-a-certain-threshold architected index. They propose a hardware lock for a pair of warps to acquire. The first warp that asks for a shared register acquires the lock and disallows the execution of its pair until it reaches the end of program. The main shortcoming of this solution is the one-time acquire with no in-kernel release. In other words, the warp that gets the ownership of the shared registers will not release it until the warp is finished. In addition, the solution requires hardware modification at the register file level for the accesses. For each register access, up to three conditions have to be verified (if the warp is shared, if the register is shared, if the lock is already acquired). The warp will have to loop inside these conditions if the answer to all of them is positive.

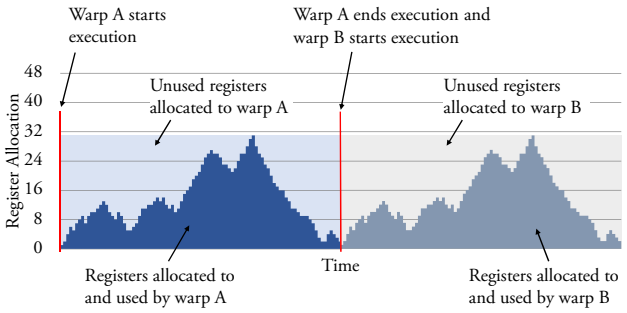
Jeon *et al.* [3] suggest virtualizing the register file to share the physical registers between the warps. They suggest embedding the dead or liveness information of the architected registers into the source code by the compiler, and using a Register Renaming Table (RRT) inside the hardware to proactively release dead registers from one warp and re-allocate them to a different warp. In other words, they borrow the idea of RRT from CPUs and implement it for the GPUs. The final outcome of this scheme is a smaller register file and reduced power consumption at the unignorable expense of higher hardware complexity. Jing *et al.* [4] take an even more drastic measure; they emulate the behavior of the register file using a cache by combining the register file and SM-private L1 cache. These solutions necessitate heavy hardware modifications such as RRT, Release Flag Cache as well as adding required support in the fetch stage of the GPU pipeline.

These drawbacks motivate the need for an inter-warp register sharing approach that introduces low hardware complexity while, simultaneously, being effective at reducing the underutilization of the register file.

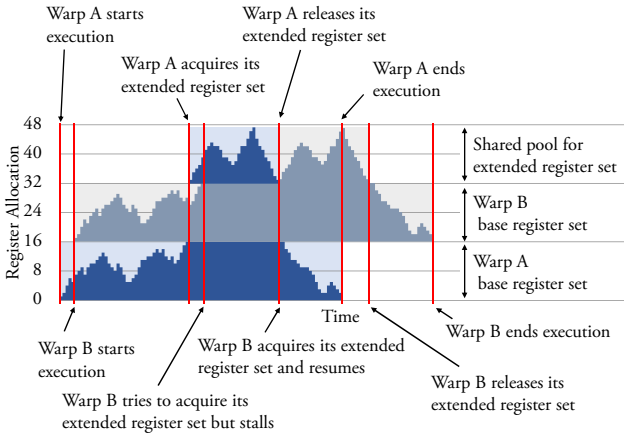
## III. RegMutex: Inter-Warp Register Time-Sharing

In this section, we propose RegMutex, an effective approach to remedy GPU register file underutilization. RegMutex time-multiplexes the allocation of a subset of registers required by the kernel between multiple warps. During the execution of the kernel, when a warp is at a program point in which it does not work with any of the registers in this subset, its execution progresses normally. However, when the warp needs this subset, those registers need to be obtained from a shared register pool for the warp. More formally, RegMutex divides the architected register set into base register set $B_s$ and extended register set $E_s$. $B_s$ is assigned to physical registers in the register file as soon as the warp resides in the SM, similar to what we observe in existing hardware. On the other hand, $E_s$ is allocated to register files only when the program requires more live registers than $|B_s|$; and also $E_s$ is de-allocated right after the number of live registers in the kernel becomes equal to or less than $|B_s|$. The communality of the shared register pool enables on-demand register allocation for segments of the GPU kernel where the number of live registers increases.

When a warp is launched for execution on the hardware, while the $B_s$ physical register assignment is instantaneous and lasts for the duration of warp execution, the allocation of $E_s$ is controlled via compiler-generated instructions that enforce an *acquire-release* semantics. The compiler identifies the code segments in the program where the number of live registers exceeds $|B_s|$. Right before entering each such code




(a) Baseline execution without RegMutex.




(b) Execution using RegMutex.

Figure 2. Example of two warps A and B executing identical code with and without RegMutex. Base register set size is 16 registers (per thread) as well as the Shared Register Pool (SRP) size. The architecture is assumed to have 48 hardware registers per thread.

segment, the compiler inserts an instruction that *acquires* $E_s$ from the Shared Register Pool (SRP). And immediately after each such code segment, the compiler inserts a *release* instruction to release its $E_s$ back to SRP. For an acquire, if the currently unused registers in SRP is insufficient to satisfy the acquire instruction for the extended registers, the warp has to wait for a release by another warp in SM to free up shared registers. In this case, the warp stalls and only becomes eligible for execution once sufficient shared registers have been freed up. In summary, the compiler drives the warps within SM to time-multiplex the pool of shared hardware registers.

Figure 2 shows a simplified, illustrative example of a case akin to typical GPU execution, where multiple warps execute the same code. Each warp has a maximum register requirement of 31 registers per thread in the example. As shown in Figure 2(a), a baseline architecture without RegMutex reserves 31 registers per thread for the full duration of the execution of each warp, preventing any overlapping of execution of the two warps (as the combined register use of the two warps, at 62, exceeds the 48 available hardware registers per thread). Figure 2(b) shows an execution configuration using RegMutex where base register sets of 16 registers per thread each and extended register sets of 16 registers per thread each are utilized by the two warps. Here, the code regions that only require the base register sets can execute in parallel, serializing only the portions that require use of the extended register sets thus enhancing the execution time. Note that in this example, for simplicity, we assume the registers are the only hardware resource constraint.

RegMutex allows a warp to acquire and release its $E_s$ as many times as needed during its lifetime. Here we enforce a fixed $|E_s|$ for the acquires within a kernel. Also, nested acquire-release instructions are not permitted. In other words, an acquire after another acquire without an intervening release or a release after another release without an intervening acquire should have no effect. Both these assumptions keep the hardware complexity of the design low and enable flexible use of acquire or release within conditional code. To facilitate the discussion on our solution in this paper, we assume that concurrent warps on an SM execute identical programs, which is the case for the majority of GPU applications.

In the rest of this section, we elaborate upon the compiler and architecture support for our technique.

### A. Compiler Support

The compiler performs a number of methodical steps to support RegMutex:

1) Register liveness analysis of the GPU assembly code to extract the register usage information.
2) Extended register set size determination.
3) Acquire/release primitive injection into the assembly code.
4) Architected register index compaction before and throughout the release state.

The first two steps analyze the kernel program and the latter two modify it. After these steps, the GPU kernel contains functionally the same code added only with extended register set acquire and release directives at proper locations. We now elaborate upon these steps.

*1) Register Liveness Analysis:* RegMutex relies on static (compile-time) register liveness information for setting the boundaries for extended register set use. Register liveness analysis helps our technique to recognize the program's register requirements at different instructions in order to instruct the executing microarchitecture for extended set acquire or release actions at appropriate locations.

As in [15], we define the static liveness for an architected register index as the set of (not necessarily consecutively placed) instructions at which the value previously written into the register has to be held intact since there is a non-zero probability that it will be read later. Figure 3 shows an example from a GPU program and the static liveness of registers. Within a basic block, if an architected register is written (*defined*) at an instruction and read (*used*) at some later instructions for the last time and without any intervening register definition, all the instructions between the definition point and the last use point are considered *live* for that

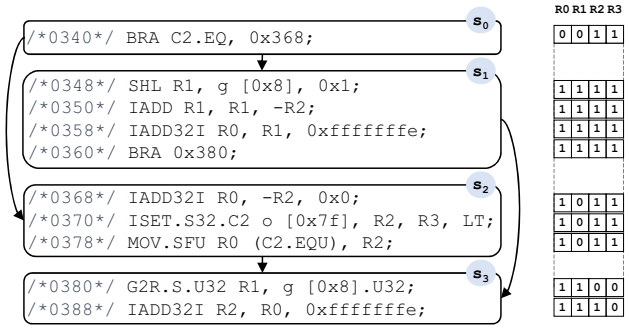



Figure 3. A GPU code sample from DWT2D application and its static register liveness.

particular register. Register R1 in basic block $s_1$ in Figure 3 is an example of this case.

However, in the presence of control flow divergence, liveness analysis is not straightforward because of the unavailability of path traversal information at compile time. If a register is defined before a branch and is used within at least one of the branched basic blocks, the register has to be considered alive within all the resulted branch basic blocks due to the uncertainty of the execution serialization by threads within the warp. In other words, the compiler has to be conservative in its assumptions. This makes the immediate post-dominator instruction of the branches the first candidate for considering the architected register *dead*. For instance, in Figure 3, although R3 is used within only $s_2$ it has to be considered alive throughout $s_1$ as well. Similarly, if a register is defined within a branch and is going to be used in a post-dominator basic block, it has to be assumed alive in other branches. The liveness status of Register R2 throughout basic block $s_1$ in Figure 3 is due to this observation.

RegMutex performs this analysis on architected registers in the GPU assembly program. The outcome of this step is a collection of Boolean vectors each representing the liveness of particular architected registers at particular instructions. We have visualized this in Figure 3 [1]. This information will be used in the next steps to determine the appropriate size for the extended register set as well as to inject the compiler-to-microarchitecture directives at appropriate program locations.

*2) Extended Register Set Size Selection:* Using static liveness information, the compiler now needs to determine the size of the extended register set, i.e., $|E_s|$. Note that $|E_s|+|B_s|$ is fixed and equals to the total number of registers the kernel asks for, hence, selecting either $|E_s|$ or $|B_s|$ enforces the value of the other. Also, note that increasing $|E_s|$ may have adversarial effects. On one hand, a large $|E_s|$ is expected to give higher occupancy by allowing more warps to be resident on the SM. On the other hand, the compiler has to mark larger sections of the program as being in an acquire state, thus warps execute more instructions while holding their extended set which may result in more contention over SRP sections during the run time.

As both the number of warps resident on SM and their scheduling freedom impact overall hardware utilization and performance, we use a simple yet methodical policy that achieves a desirable trade-off between improving physical register utilization and not curtailing scheduling freedom. After finding the baseline kernel's theoretical occupancy and the contribution of kernel register usage as a limiting factor, we select candidate values for $|E_s|$ from an empirically-derived set of $\{0.1, 0.15, 0.2, 0.25, 0.3, 0.35\}$ multiplied by the number of registers used by the kernel. Then we keep the even numbers that result in the highest occupancy calculated only with the base set size. If multiple candidate elements for $|E_s|$ give the same theoretical occupancy, we go with the largest element that possibly results in concurrent progress of more than half the warps in the current occupancy in the acquire mode. Let us illustrate these steps using an example. Assume the kernel asks for 24 registers to run on our baseline (Nvidia Fermi architecture) which supports up to 20 registers per thread without limiting the occupancy. Let us also assume that register usage is the only factor limiting the theoretical occupancy here. Based on our approach, the candidate set for $|E_s|$ consists of the even numbers in $\lfloor 24 \odot \{0.1, 0.15, 0.2, 0.25, 0.3, 0.35\} \rfloor$ ($\odot$ is the element-wise product) which yields $\{2, 4, 6, 8\}$. From this set, 4, 6, and 8 result in $|B_s|$ equal to 20, 18, and 16 respectively which have full occupancy for only the base set. Assuming maximum number of warps per SM is 48 and the total number of registers in the register file is $32K$, these configurations leave 16, 26, and 32 sections for SRP which indicate the number of warps that can acquire $E_s$ concurrently. RegMutex selects $|E_s| = 6$ since it is the largest candidate that allows more than half of the warps on the SM (in this example 26) in the calculated occupancy be in the acquire state.

After determining the number of registers in the extended set, the new theoretical occupancy of the kernel is obtained using calculated $|B_s|$. This occupancy gives the number of CTAs that the SM can host, and also determines the total size for the SRP.

*Deadlock Avoidance:* To avoid deadlocks in our design, two additional rules govern $|E_s|$ selection. First, the distribution of $|B_s|$ and $|E_s|$ has to be such that there are enough registers in the shared pool for at least one warp's $E_s$. This ensures that warps do not stall indefinitely for an acquire. Second, $|B_s|$ has to be greater than or equal to the number of live registers at any point in the program that CTA-wide synchronization primitives such as __syncthreads() exist. This avoids any deadlock due to inter-dependency of warps. In other words, while a warp $W_a$ is waiting for another warp $W_b$ to arrive at the synchronization PC, warp $W_b$ will not wait at an acquire instruction for warp $W_a$ to release its extended register set.

[1] A similar analysis and liveness representation are provided for CUDA application by the *nvdisasm* CUDA binary tool [16].

*3) Acquire/Release Primitive Injection:* After recognizing the regions within the program that use the extended register set, the compiler injects acquire and release primitives respectively at the beginning and the end of such regions. For RegMutex, we create an instruction to convey acquire or release information to the hardware. Unlike [3] RegMutex need not rely on meta-instructions since the content of this instruction is either a release or an acquire command.

*4) Architected Register Index Compaction:* To preserve the simple $Y = X + B$ equation for the architected to physical register assignment (different $B$'s for extended and base register sets), architected register indices have to stay within their boundaries during the release state. The compiler must therefore ensure that none of the extended register set members contain live values when the extended register set is released. We essentially need a mechanism to compact the architected register indices for the duration the extended set is not acquired, and also right before releasing the extended register set.

To achieve this goal, the compiler may have to move any live values in the extended register set to available registers in the base set during the release state and before releasing the extended register set. For example, let us assume a scenario where the base register set size is 6 and a warp has a live register set $\{2, 4, 5, 9\}$ right before the release. Before releasing the extended set, the compiler has to move architected register 9 to one of 0, 1, or 3 locations. Note that the movement of architected registers is instrumented by the compiler (usually with `MOV` operations). This is similar to what [7] suggests under the name of *register declaration reordering* but it is different in that index compaction may happen multiple times right before each release by moving the architected registers, whereas in *register declaration reordering* the index minimization is limited to happen only once by reordering register declarations. In addition, the compiler has to apply register location renaming for all the uses of that particular register until the end of its current live range. We use a similar analysis to that done in Section III-A1 for those registers exceeding the boundary at release states.

We emphasize that the above compiler analysis steps, when embedded within a compiler, need to be applied during the last stages of the compilation chain. This is because the technique needs to know the architected register assignment whereas compiler middle-ends such as LLVM work with virtual registers in SSA form.

### B. Architecture Support

In this section, we explain the architectural requirements to enable RegMutex. We used the baseline design offered by *GPGPU-sim* [10], a simplified depiction of which is shown at the top of Figure 4. After decoding acquire/release primitive at the decode stage as a barrier operation, the acquire or releases command is given to the issue stage. At this stage, the warp acquires the extended set or waits for an extended

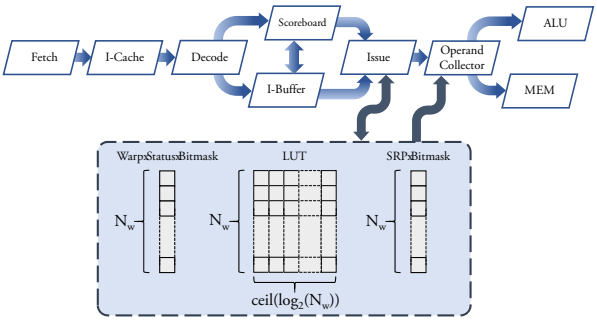
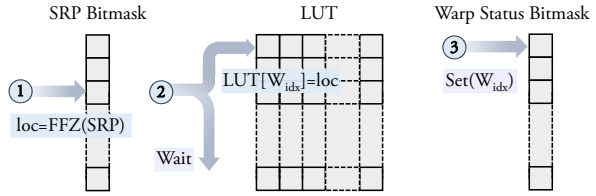



Figure 4. The baseline design from *GPGPU-sim* [10] (top) and RegMutex's added storage structures (bottom). Specified sizes are in bits.

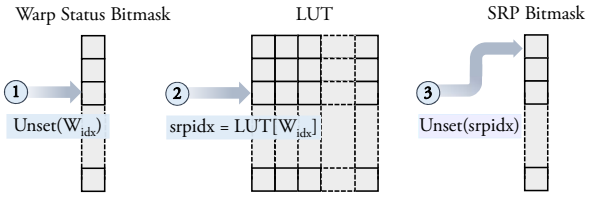
set to be freed, or releases its extended set. Upon a successful acquire, the information for the acquired SRP section as the extended set is passed to the Operand Collector Unit for register mapping. Below we elaborate upon the hardware implementation of RegMutex at these two stages.

*1) Warp Issue Management Organization:* In RegMutex a warp has to ask for physical registers for its extended set upon reaching an *acquire* instruction. If no physical extended set is available, the warp has to wait for another warp to *release* their set. The warp needs to essentially imitate the behavior of already existing barrier synchronization and communication in GPUs. CUDA barrier synchronization instructions allow one warp $W_a$ to signal its arrival to another warp $W_b$, and to warp $W_b$ to wait for warp $W_a$ to arrive at a particular barrier. These instructions have been used for warp specialization purposes and implementing producer-consumer models [17], [18] via PTX instructions `bar.sync` and `bar.arrive`. For RegMutex, we exploit a similar design where warps have to wait for the release of an extended physical set when faced with *acquire* or signal the release of their own extended physical set upon a *release*. Since barrier operations in SM are executed at the issue stage, we design RegMutex's allocation logic closely coupled with it.

The highlighted section of Figure 4 shows the RegMutex's modification interacting mostly with the issue stage of the microarchitecture model. Because we need to keep track of each warp's execution mode (*acquired* or *not-acquired*), we use a single bit per warp to indicate the warp status. In the baseline model, each SM can host up to 48 resident warps ($N_w = 48$) and therefore the warp status bitmask is 48 bits long. This bitmask is indexed by the warp index within the SM. In addition, another bitmask holds the status of sections of the Shared Register Pool (SRP). Since there can be up to $N_w$ sections in the SRP, and since we disallow nested acquires and releases, the SRP bitmask is $N_w$ bits long as well. Each bit in SRP bitmask indicates if a particular extended physical register set is acquired or not. The mapping between a warp and a bit in the SRP bitmask is performed via a lookup table (LUT in Figure 4). The table has one entry for each warp while each entry contains $\lceil \log_2 N_w \rceil$ bits indicating which one of the $N_w$ SRP sections the warp




(a) Acquire.




(b) Release.

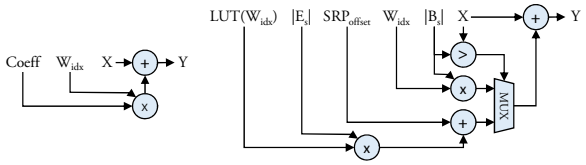
Figure 5. Acquire/release procedure implementation in RegMutex.

has acquired (if warp's status bit is set). The total size for this table in our baseline is therefore $48 \times 6 = 288$ bits.

As depicted in Figure 5(a), when an acquire instruction reaches the issue stage, SRP bitmask is searched for an unset (zero) bit. This is equivalent of the Find First Zero (FFZ) operation on the SRP bitmask which returns the index of the least significant zero bit. If a valid index (for instance, $0 <= idx < 48$ in the baseline model) is returned, a section is available. Therefore, the index is written into the lookup table and the warp's status bit and the SRP availability bit are set. This index is then passed to the Operand Collector Unit. However, if the returned index is invalid, the warp waits at the barrier and retries at later rounds when the warp gets scheduled again. Moreover, when a release instruction arrives at the issue stage, the warp status bit is unset, and the warp's acquired SRP section index is retrieved from the lookup table, as shown in Figure 5(b). This index determines the bit to unset in the SRP bitmask, specifying the release of the previously acquired extended physical register set. Also, note that in case the extended register set size does not allow having maximum number of SRP sections, those bits in SRP bitmask that do not correspond to any SRP section are set at the beginning of the kernel placement and stay intact throughout the execution.

Total number of bits introduced into the baseline by RegMutex is 384. Compared to register file virtualization approach [3], which requires $30,240$ bits for the renaming table and 1024 bits for register availability indication (excluding Release Flag Cache), RegMutex reduces the additional structure storage cost by more than 81x. Moreover, since the introduced acquire/release instruction is simple, RegMutex does not need to use meta-instructions, as opposed to [3], which necessitates partitioning the fetch stage into two separate stages.

*2) Architected-To-Physical Register Mapping:* In GPUs registers are allocated per warp and indexed by the warp ID within SM. For instance, the baseline design from *GPGPU-sim* [10] is from Nvidia Fermi architecture containing 32K 32-bit physical registers. Given there are 32 threads within the warp, there are 1K of physical register *packs* to distribute among the warps. As we mentioned earlier, to map packs of architected registers to physical registers GPUs use a simple equation $Y = X + B$. In this equation, $B$ is a warp specific base address assigned at run time and is resulted from multiplying the warp index within SM ($W_{idx}$) with a constant coefficient ($Coeff$) determined by the kernel's total register usage: $B = Coeff \times W_{idx}$. This baseline design can be viewed in Figure 6(a) and is implemented within GPU's Operand Collector unit before accessing the register file banks.




(a) Baseline. (b) Augmented for RegMutex.

Figure 6. Architected to physical register mapping design in the Operand Collector Unit. $X$ is the architected register index and $Y$ is the resulted physical register index.

To support RegMutex, we augment the baseline design as shown in Figure 6(b). Since the base addresses for physical registers designated to hold $B_s$ and $E_s$ of a warp are disjoint, the warp compares the architected register index with $|B_s|$ to realize if the register belongs to the base set or the extended set. If the register belongs to the base set, in a fashion similar to baseline, the warp index within SM gets multiplied to $|B_s|$ to result the base address for the physical register. Otherwise, the SRP section assigned to the warp ($LUT(W_{idx})$) is multiplied by $|E_s|$ to get the base address within SRP. The result is added with $SRP_{offset}$, the offset of SRP within register file, to constitute the physical base address for the register. In this design, the values of $|B_s|$ and $|E_s|$ are supplied by the compiler and alongside $SRP_{offset}$ are given to the Operand Collector Unit at the kernel launch.

In summary, RegMutex enjoys a much simpler design compared to existing approaches such as [3] and [5] which micromanage the allocation of every register and necessitate additional structures such as Release Flag Cache.

### C. Paired-Warps Specialization

In this part, we introduce a specialization of RegMutex that, rather than time-multiplexing the registers across all the SM's resident warps, shares the extended register set between specific pairs of warp. Although this approach reduces the register sharing opportunity, it lowers the amount of hardware modifications even further. In this specialization, the design sets aside $2 \times |B_s| + |E_s|$ physical registers for each pair of warps. Each warp's $|B_s|$ allocation is static and exclusive but $E_s$ is time-multiplexed between the two. Therefore, while both warps can move forward in release state, only one

of them may progress in the acquire state, disallowing the other warp to acquire $E_s$ until the release point. Paired-warps specialization of RegMutex eliminates the need for the lookup table and the SRP bitmask, and only requires a bitmask with the length half the maximum number of warps in the SM, i.e., $N_w/2$, to specify the status of the extended set shared between pairs of warps.

## IV. Experimental Evaluation

To evaluate RegMutex's performance, we extended *GPGPU-Sim v 3.2.2* [10] simulator. We used the microarchitecture specifications for GeForce GTX480 GPU that comes with the simulation framework. It includes 15 SMs, 128 KB register file size per SM, 2 warp schedulers per SM, and the default greedy-then-oldest scheduling policy. In addition to the register file size in SM, we allow shared memory usage per SM and the maximum number of resident threads in SM to act as other constraints that affect the theoretical occupancy of the CUDA kernels. Note that although we carry out our simulations based on an Nvidia Fermi GPU, the principles behind register allocation on newer Nvidia CUDA-enabled architectures including Kepler, Maxwell, Pascal, and Volta have stayed the same: registers are still statically and exclusively reserved. Therefore, the resulting register file underutilization challenge does indeed still exist. Even though per-SM register file size has been doubled in newer architectures, the maximum number of resident warps on the SM in newer GPUs is also increased. As a result, in all post-Fermi Nvidia GPUs having more than 32 registers per thread definitely results in incomplete occupancy, which is troubling for applications with high register demand. Hence, our solution is applicable and generalizable to newer GPU architectures as well.

We utilized PTXPlus to extract basic block information as well as control flow analysis in order to implement RegMutex's compiler support. PTXPlus is a tool integrated with GPGPU-Sim that enables implementation of compiler optimizations when working with the simulator. It uses an augmented form of PTX intermediate representation that is extracted from the binary, and therefore is expected to fully preserve the optimizations applied at the PTX-to-SASS level. PTXPlus is the closest level to machine code that GPGPU-Sim allows for applying compiler optimizations.

We selected a total of 16 applications from Rodinia [11], Parboil Benchmark Suite [12], and Nvidia CUDA SDK [19] to verify the effectiveness of RegMutex under different workloads. These applications are shown in Table I and exhibit different SM resource requirements. Please note that these workloads suffer from high register usage and are selected to show the benefits of RegMutex in different scenarios. Applications that do not have such property are not affected by applying our technique since RegMutex evaluates all the registers as the members of the base register set, therefore, it does not insert any acquire or release instructions into the program. Moreover, none of the presented workloads incur simultaneously executing dissimilar kernels. Co-scheduling dissimilar kernels on an SM is not supported by our technique and results in falling back to the default execution mode (zero-sized extended set). Table I also specifies the number of registers per thread for each kernel. The numbers in the parenthesis show the number of registers rounded to the upper multiple of 4. The simulation framework uses this number for resource allocation calculations. We also showed the calculated base set size for RegMutex for each application in the table. All the applications are compiled with NVCC 4.0 and GCC 4.6 with -O3 compilation flag. Since PTXPlus is not compatible with CUDA Compute Capability 2.0 or higher, applications are compiled for Compute Capability 1.3.

| Application | # Regs. | $\lvert B_s \rvert$ | Application | # Regs. | $\lvert B_s \rvert$ |
|---|---|---|---|---|---|
| BFS | 21 (24) | 18 | Gaussian | 12 (12) | 8 |
| CUTCP | 25 (28) | 20 | HeatWall | 28 (28) | 20 |
| DWT2D | 44 (44) | 38 | LavaMD | 37 (40) | 28 |
| HotSpot3D | 32 (32) | 24 | MergeSort | 15 (16) | 12 |
| MRI-Q | 21 (24) | 18 | MonteCarlo | 13 (16) | 12 |
| ParticleFilter | 32 (32) | 20 | SPMV | 16 (16) | 12 |
| RadixSort | 33 (36) | 30 | SRAD | 18 (20) | 12 |
| SAD | 30 (32) | 20 | TPACF | 28 (28) | 20 |

Table I
Workloads used in experiments. The number of registers per thread and RegMutex's base register set size are shown for each kernel.

### A. Kernel Occupancy Boost Analysis

We first analyze the performance improvement enabled by RegMutex for 8 GPU kernels from Table I on the baseline architecture. The theoretical occupancy of this set of kernels are limited by the excessive register demand, hence, enhancing their occupancy by time-sharing a portion of the registers using RegMutex can be beneficial. For these applications, Figure 7 shows the percentage of execution cycle reduction with RegMutex calculated with respect to the number of baseline execution cycles. It also shows the influence of RegMutex on the theoretical occupancy of the kernel by plotting the initial occupancy of the kernels alongside the occupancy with RegMutex. On average, RegMutex has reduced the execution cycle of the kernels by $13\%$ via enhancing the overall register file utilization.

In a case such as BFS, the boost in the occupancy resulted in $23\%$ reduction in the execution cycles. On the other hand, SAD application does not enjoy such performance improvement with the same amount of occupancy enhancement. This tells us that theoretical occupancy cannot be directly indicative of the performance enabled by RegMutex, yet, is one of the contributing factors. In the case of BFS and SAD, extended set size and SRP section are other impactful parameters. SAD requires a considerably larger extended set compared to BFS (see Table I) for the occupancy increase. This makes the number of SRP sections limited which increases the contention over acquiring the extended

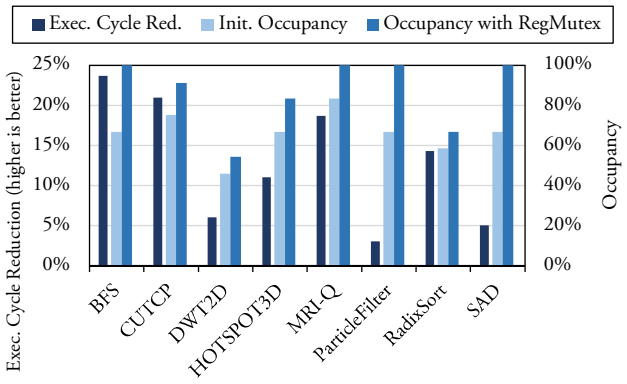



Figure 7. The performance improvement enabled by RegMutex over the baseline.

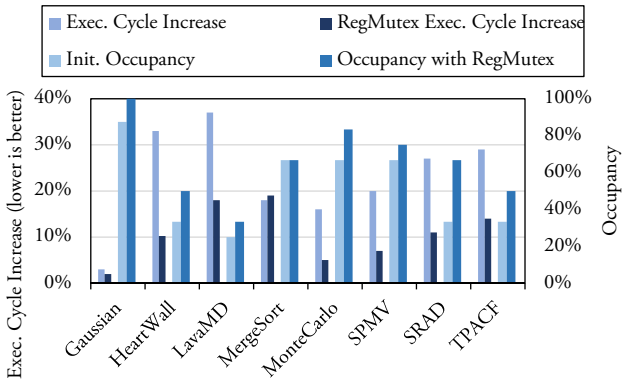



Figure 8. The performance of applications with and without RegMutex on an architecture with half the baseline's register file size.

set between warps. DWT2D and ParticleFilter applications suffer from the same issue as well. Program nature is another contributing factor to the performance improvement provided by RegMutex. A kernel that holds an extended set more often and for longer instructions increases the chance of other warps having to wait at acquire points. The contribution extent for all these parameters depend on each other, and most importantly, for typical kernels that are data-driven, to the input of the kernel. Therefore, speculating suitable parameters using heuristics requires careful analysis of the program.

### B. Register File Size Reduction Analysis

In this part, we analyze the effect of RegMutex on 8 applications for which the register file size is not limiting the theoretical occupancy. For these applications, similar to GPU-Shrink [3], we halve the register file size of the baseline to 64 KB per SM and see the effect of this reduction with and without RegMutex and compare it with the baseline mode. Jeon *et al.* [3] argue that this design leads to significant power savings by reducing the register file dynamic and overall power consumption by 20% and 30% respectively. Note that unlike GPU-Shrink we did not enforce register spilling, but rather allowed GPU to determine the number of resident warps on SM under specified circumstances.

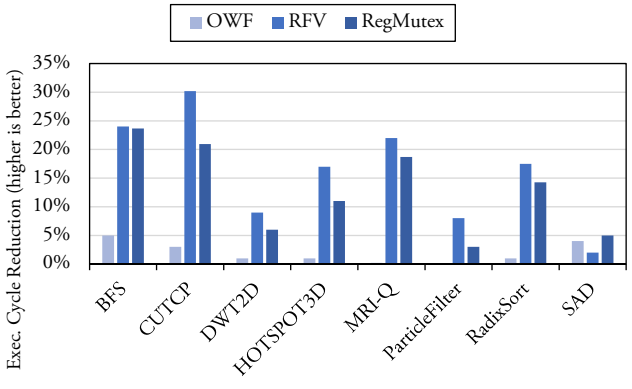
Figure 8 compares the execution cycle for scenarios where RegMutex is present and absent with the kernel's number of execution cycles for the architecture with full register file. It is evident that in presence of RegMutex, the kernel experiences much less increase in the number of execution cycles and allows the kernels to preserve the performance when an architecture with smaller number of physical registers is provided. For the applications in Figure 8, while the design without RegMutex suffers from 23% increase in the number of execution cycles on average, with RegMutex we observe an average of 9% growth in the number of execution cycles.

We also plotted the occupancy of the kernels before and after applying RegMutex on the architecture with half the baseline's physical registers in Figure 8. Similar to previous part, we observe that occupancy is a contributing factor to RegMutex's performance. In 7 out of these 8 applications RegMutex has successfully increased the register utilization by enhancing the occupancy of the kernel. It is only in MergeSort workload that our heuristic for extended register set size determination comes up with a size that does not increase the occupancy. Therefore, we observed no benefit, but a slight increase in execution cycle due to added RegMutex instructions. This is, in fact, the only workload among 16 applications for which the default RegMutex incurred slowdown.

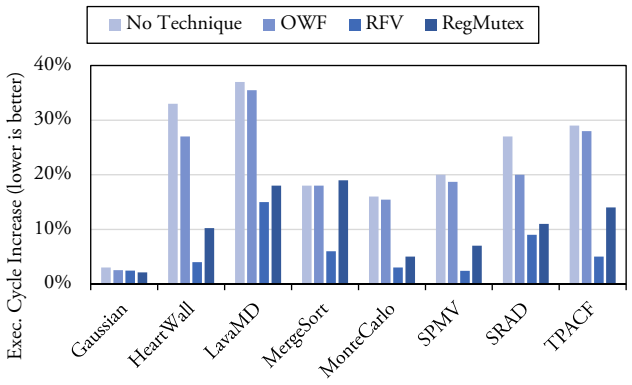
### C. Performance Comparison with Related Work

In this section, we compare performance improvement provided by RegMutex with the two most closely related approaches: i) Resource Sharing with the *Owner Warp First* (OWF) scheduling optimization [7], and ii) *Register File Virtualization* (RFV) [3]. To perform an apples-to-apples comparison, we used implementations of both approaches on GPGPU-Sim similar to the extensions used to implement RegMutex. Figure 9(a) presents the results of the comparison on the baseline architecture. The average *reduction* in kernel execution time in cycles is 1.9%, 16.2%, and 12.8% for OWF, RFV, and RegMutex respectively. We see that both RFV and RegMutex significantly out-perform OWF. While the improvement due to RFV is 3.4% higher than that of RegMutex on average, RegMutex has much lower hardware implementation complexity than RFV, as discussed in Section III-B. RVF demands more than 31 kilobits for additional structure storage in the default architecture with 128 KB registers, whereas RegMutex only needs 384 bits, reducing the storage requirement by more than 81x.

We also perform this comparison on the architecture with half the baseline's register file size. The results are shown in Figure 9(b). We observe an average of 22.9% *increase* in execution cycles that results from halving the register file size when no technique is applied. The average increase in kernel execution cycles is 20.6%, 5.9%, and 10.8% for OWF, RFV, and RegMutex respectively. Again, we see that both RFV




(a) On the baseline architecture.




(b) With half the baseline architecture's registers.

Figure 9. RegMutex performance comparison with Register File Virtualization (RFV) [3] and the work of Jatala *et al.* [7], which we refer to it as OWF.

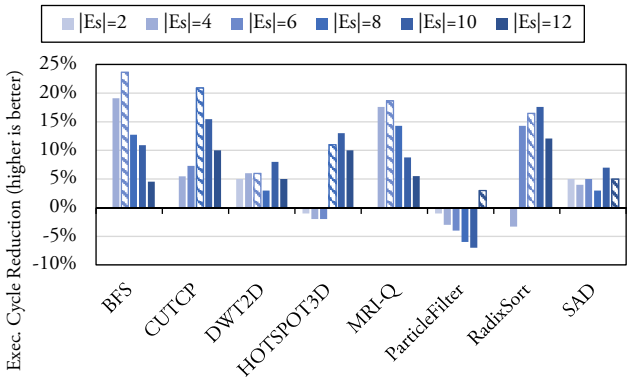
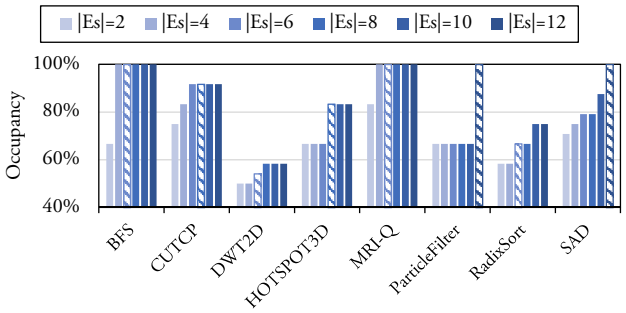



Figure 10. The sensitivity of kernel performance to variations in the extended set size with RegMutex. Columns with diagonal stripes are our heuristic's pick.

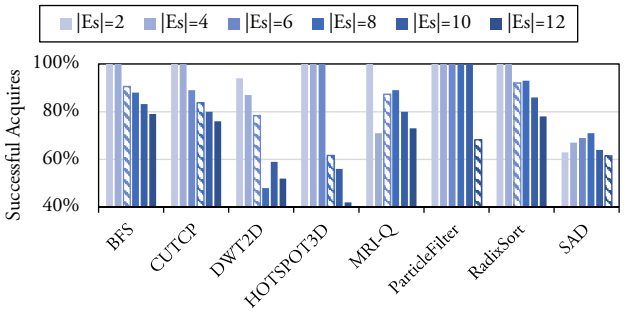
and RegMutex significantly out-perform OWF in this case as well, and that RFV performs better than RegMutex, but does so with increased hardware implementation complexity (as discussed in Section III-B).

### D. Extended Set Size Sensitivity Analysis

Here we analyze the performance sensitivity of our technique to the size of the extended set. In Section III-A2




(a) Theoretical occupancy.



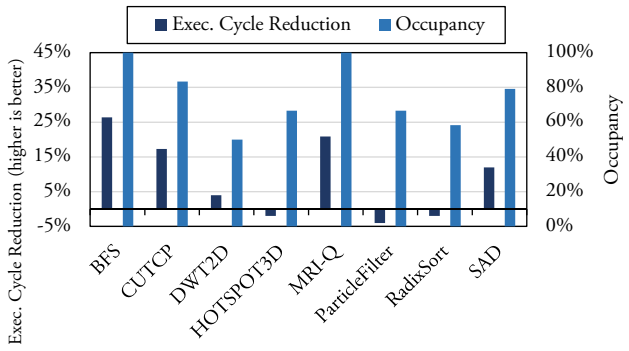

(b) Successful acquires among all acquire instructions.

Figure 11. The variations in the theoretical kernel occupancy and the ratio of successful acquires with respect to changes in the extended set size. Columns with diagonal stripes are our heuristic's pick.

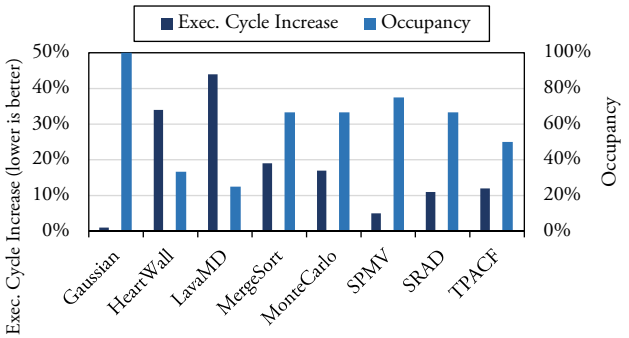
we mentioned that the size of the extended set, i.e., $|E_s|$, which is chosen at compile time, affects the performance in two ways. Increasing $|E_s|$ results in $|B_s|$ decreasing which allows more concurrent warps to reside on the SM thus enhancing the occupancy. On the other hand, a higher $|E_s|$ means larger sections of a program are marked as acquire state therefore it is more probable for warps to be holding extended sets, hence, warps may have to wait more often and for longer times before they can acquire physical registers for their extended set.

To observe the influence of extended set size on the performance of kernels, we manually set $|E_s|$ to 2, 4, 6, 8, 10, and 12, and observed the execution cycle reductions. Figure 10 plots the results. We distinguished the extended set size determined by our heuristic (described in Section III-A2) using diagonal stripes. As you can see, although the best performing $|E_s|$ differs from one application to another and does not follow any particular trend, our method has been able to pick the best or one of the best extended set sizes for each application. This is due to prioritizing occupancy and then adjusting it based on the number of sections in SRP.

To further investigate the results, for different $|E_s|$'s, we measured the theoretical occupancy of each kernel and the percentage of successful acquire requests with respect to all acquire instructions executed and plotted them in Figure 11(a) and Figure 11(b). By comparing the results in these plots, it becomes clear that as $|E_s|$ gets larger, occupancy increases




(a) Execution cycle reduction is measured against the baseline architecture.




(b) Execution cycle is measured for the architecture with half the baseline physical registers. To be consistent with previous plots, the increase is measured against the baseline kernel performance on the architecture with full register file.

Figure 12. The effect of RegMutex's paired-warps specialization on the execution cycle and the occupancy of kernels.

but the chance of a successful acquire usually reduces. Both of these two adversarial effects contribute to RegMutex's performance. This makes suitable $|E_s|$ selection a challenging task that requires careful observation of the program behavior as well as static calculation of the kernel occupancy in the given architecture.

### *E. Paired-Warps Specialization Performance Analysis*

As we mentioned earlier in Section III-C, paired-warps specialization of RegMutex eliminates the need for the SRP bitmask and the lookup table by privatizing SRP sections among pairs of warps. This reduces the hardware storage cost by more than 20x compared to the non-specialized RegMutex at the expense of lower generality. Figure 12(a) shows the execution cycle reduction and the resulted theoretical occupancy after applying paired-warps RegMutex to the baseline architecture. As can be derived by comparing this figure with Figure 7, this specialization is effective when the occupancy can be improved, as is the case for 5 of our 8 applications. For a few applications such as SAD, we observe even a higher reduction in execution cycles compared to the default RegMutex. We found that this is generally due to higher probability of acquires with this

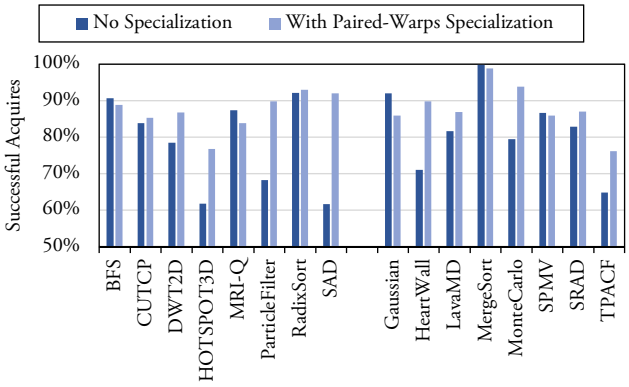



Figure 13. Acquire instruction success rate in RegMutex with and without paired-warps specialization. The results for the 8 leftmost applications are reported on the baseline architecture, and the rest, on the architecture with half of the baseline register file size.

specialization, as shown in Figure 13. While paired-warps specialization guarantees the exclusive access to the extended set for a warp be shared with at most one other warp, default RegMutex may have to share a few SRP sections among many resident warps on SM. This can lead to an increased waiting time on acquire instructions for the default mode. However, when the occupancy stays intact, paired-warps specialization is unable to improve the performance. The inability to improve the occupancy in such cases stems from the guarantee this specialization has to provide for pairs of warps. This makes paired-warps specialization susceptible for such scenarios while the default mode exhibits flexibility and therefore resistance in these cases. In other words, exclusivity of $2 \times |B_s| + |E_s|$ registers for pairs of warps makes the specialization outperform or underperform the default mode depending on the application. On average, paired-warps specialization reduces the execution cycles for applications in Figure 12(a) by $8\%$ which is $4\%$ less compared to the default mode.

Figure 12(b) illustrates the increase in the execution cycles as well as the resulted occupancy when paired-warps specialization is used on the architecture with half the baseline physical registers. Here we observe the similar phenomenon as we described above as well. When the occupancy stays the same, as it is the case for 4 out of 8 applications, no performance improvement is provided. However, in other cases where the occupancy could be increased, paired-warps specialization becomes effective. For these applications, this specialization has increased the execution cycles over the baseline with full register file by $17\%$ which is $5\%$ less than the baseline with half the register file but it is outperformed by the default RegMutex by $8\%$ difference.

## V. Related Work

As we explained earlier in Section II, a number of directly related works propose solutions for static and exclusive GPU register assignment. Tarjan *et al.* [5] suggest virtualizing

the registers and assigning them onto physical registers on-demand. However, this method is expected to incur hardware complexities even beyond what was proposed by Jeon *et al.* [3]. Gebhart *et al.* [6], [20] propose multi-level register file designs where long-lived registers and short-lived registers are placed in different register hierarchies for power efficiency purposes. While imposing high amount of modifications to the existing hardware, these solutions also lack generality due to the fixed sizes of the register file hierarchy levels. RegMutex, in contrast, does not disturb the performance of an application that does not utilize it. Tan and Fu [21] suggest another hierarchical approach where registers are classified into *fast* and *slow* categories to reduce susceptibility of GPU register file to process variation. However, RegMutex's aim is to offer approximately the same performance at a lower cost, or higher performance at the same cost, by reducing the number of required registers or by allowing residence of more warps on an SM. Also, as opposed to the work by Jatala *et al.* [7], RegMutex offloads the register ownership arbitration to the compiler and allows the set of shared registers be handed over between the warps multiple times. Zorua [22] is another work that utilizes a runtime-compiler-hardware synergistic approach for resource virtualization at runtime. While in Zorua, performance portability across multiple architectures is the goal and is achieved via virtualizing on-chip resources, RegMutex tackles the challenge of static resource assignment *during* the kernel execution. RegLess [23] replaces register file with a smaller actively-managed staging unit and LTRF [24] suggests a hierarchical RF design, both utilizing the compiler to provide hints to the hardware for the run-time use. Unlike these works, RegMutex does not fundamentally change or replace the RF structure, and can easily be disabled or enabled by the compiler. A patent application by Coon and Lindholm [25] also has the notion of grouping threads together based on resource sharing; however, they only allow one thread from each group to execute at a time while our goal is to maximize concurrent execution while sharing a limited resource.

Another body of papers, orthogonal to RegMutex, target economical use of GPU's available resources. Kim *et al.* [26] utilized unused registers for executing the warps in a special mode called pre-execution in order to cope with long stalls due to memory accesses. Warped-Compression [27] exploits the similarity of the register values between threads within a warp during the execution in order to eliminate redundant register file occupations. Compressing similar registers into one register essentially results in saving on the GPU register file power consumption. It also resembles the works of Jourdan *et al.* [28] for CPUs where logical registers are mapped into physical ones when sharing the same content. KernelMerge [29] aims to allow co-residency of two GPU kernels on one device to enables utilization of resources that are left unutilized when only one of the kernels is running. CCC [30] utilizes on-chip shared memory to collect tasks for future warp-efficient use. Also, Yoon *et al.* [31] propose an architecture to increase the on-chip resource utilization by improving the CTA scheduling policy. While sharing the same general goal with RegMutex, we observe no restriction on simultaneous application of these works with our proposed technique.

In the CPU realm, hardware-only approaches [32], [33] as well as combined compiler-microarchitecture solutions [34], [35], [36] have been proposed for quick dead register identification and release. Ayala *et al.* [37] suggest a software-hardware technique that tags sections of the program which require only a small amount of registers for execution and allows disabling regions of the register file during the execution for energy saving purposes. As we mentioned before, such techniques in massively multi-threaded devices such as GPUs are often impractical due to the their heavy reliance on TLP between resident warps.

## VI. Conclusion

Static and exclusive register allocation on GPUs leads to register file underutilization. In this work, we addressed this challenge by introducing RegMutex, an effective synergistic compiler-microarchitecture mechanism to time-multiplex the register use between warps. On the compiler side, RegMutex divides the architected register set into a base register set and an extended register set, and by analyzing the program, injects instructions in the kernel code where the extended register set activates and deactivates. On the microarchitectural side, while physical registers are allocated to the base architected registers for the lifetime of the kernel, RegMutex takes a communal approach on allocating physical registers to the extended architected register set. Using the information provided by the compiler, the warp acquires the physical registers for extended architected registers from a shared register pool when needed, and releases them to the shared pool upon deactivation of the extended register set. We showed that this approach enhances the performance of GPU kernels exhibiting a limited occupancy due to high register pressure, and allows application resilience when underlying microarchitecture employs a smaller register file. Our experiments show that RegMutex reduces the number of kernel execution cycles by up to 23% through increasing the overall register utilization.

## Acknowledgment

We would like to thank Sudhakar Yalamanchili and Sana Damani for giving constructive feedback prior to submission. We thank anonymous reviewers for providing useful comments. This work is supported by National Science Foundation collaborative Grants No. 1629564 and 1629459.

## REFERENCES


[1] 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 *PACT*, 2016.

[2] 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 *ISCA*, 2013.

[3] H. Jeon, G. S. Ravi, N. S. Kim, and M. Annavaram, "Gpu register file virtualization," in *MICRO*, 2015.

[4] 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 *MICRO*, 2016.

[5] D. Tarjan and K. Skadron, "On demand register allocation and deallocation for a multithreaded processor," Jun. 30 2011, uS Patent App. 12/649,238. [Online]. Available: https://www.google.com/patents/US20110161616

[6] M. Gebhart, S. W. Keckler, and W. J. Dally, "A compile-time managed multi-level register file hierarchy," in *MICRO*, 2011.

[7] V. Jatala, J. Anantpur, and A. Karkare, "Improving gpu performance through resource sharing," in *HPDC*, 2016.

[8] N. Jayasena, M. Erez, J. H. Ahn, and W. J. Dally, "Stream register files with indexed access," in *HPCA*, 2004.

[9] H. Kim, R. Vuduc, S. Baghsorkhi, J. Choi, and W.-m. Hwu, "Performance analysis and tuning for general purpose graphics processing units (gpgpu)," *Synthesis Lectures on Computer Architecture*, vol. 7, no. 2, pp. 1–96, 2012.

[10] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *ISPASS*, 2009.

[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 *IISWC*, 2009.

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

[13] "Nvidia tesla v100 gpu architecture whitepaper," http://www.nvidia.com/object/volta-architecture-whitepaper.html, accessed: 2017-08-11.

[14] V. Volkov, "Better performance at lower occupancy," in *Proceedings of the GPU technology conference, GTC*, vol. 10, 2010.

[15] K. Cooper and L. Torczon, *Engineering a compiler*. Elsevier, 2011.

[16] "*nvdisasm* cuda binary tool," http://docs.nvidia.com/cuda/cuda-binary-utilities/#nvdisasm, accessed: 2017-08-11.

[17] M. Bauer, H. Cook, and B. Khailany, "Cudadma: Optimizing gpu memory bandwidth via warp specialization," in *SC*, 2011.

[18] M. Bauer, S. Treichler, and A. Aiken, "Singe: Leveraging warp specialization for high performance on gpus," in *PPoPP*, 2014.

[19] "Cuda computing sdk 4.2," https://developer.nvidia.com/cuda-toolkit-42-archive, accessed: 2017-08-11.

[20] 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 *ISCA*, 2011.

[21] J. Tan and X. Fu, "Mitigating the susceptibility of gpgpus register file to process variations," in *IPDPS*, 2015.

[22] 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 *MICRO*, 2016.

[23] J. Kloosterman, J. Beaumont, D. A. Jamshidi, J. Bailey, T. Mudge, and S. Mahlke, "Regless: Just-in-time operand staging for gpus," in *MICRO*, 2017.

[24] M. Sadrosadati, A. Mirhosseini, S. B. Ehsani, H. Sarbazi-Azad, M. Drumond, B. Falsafi, R. Ausavarungnirun, and O. Mutlu, "Ltrf: Enabling high-capacity register files for gpus via hardware/software cooperative register prefetching," in *ASPLOS*, 2018.

[25] B. Coon and J. Lindholm, "System and method for grouping execution threads," Jul. 21 2007. [Online]. Available: https://www.google.com/patents/US20070143582

[26] K. Kim, S. Lee, M. K. Yoon, G. Koo, W. W. Ro, and M. Annavaram, "Warped-preexecution: A gpu pre-execution approach for improving latency hiding," in *HPCA*, 2016.

[27] S. Lee, K. Kim, G. Koo, H. Jeon, W. W. Ro, and M. Annavaram, "Warped-compression: Enabling power efficient gpus through register compression," in *ISCA*, 2015.

[28] S. Jourdan, R. Ronen, M. Bekerman, B. Shomar, and A. Yoaz, "A novel renaming scheme to exploit value temporal locality through physical register reuse and unification," in *MICRO*, 1998.

[29] 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.

[30] F. Khorasani, R. Gupta, and L. N. Bhuyan, "Efficient warp execution in presence of divergence with collaborative context collection," in *MICRO*, 2015.

[31] M. K. Yoon, K. Kim, S. Lee, W. W. Ro, and M. Annavaram, "Virtual thread: Maximizing thread-level parallelism beyond gpu scheduling limit," in *ISCA*, 2016.

[32] M. Moudgill, K. Pingali, and S. Vassiliadis, "Register renaming and dynamic speculation: An alternative approach," in *MICRO*, 1993.

[33] J. F. Martinez, J. Renau, M. C. Huang, and M. Prvulovic, "Cherry: Checkpointed early resource recycling in out-of-order microprocessors," in *MICRO*, 2002.

[34] M. M. Martin, A. Roth, and C. N. Fischer, "Exploiting dead value information," in *MICRO*, 1997.

[35] J. L. Lo, S. S. Parekh, S. J. Eggers, H. M. Levy, and D. M. Tullsen, "Software-directed register deallocation for simultaneous multithreaded processors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 9, 1999.

[36] T. M. Jones, M. F. R. O'Boyle, J. Abella, A. Gonzalez, and O. Ergin, "Compiler directed early register release," in *PACT*, 2005.

[37] J. L. Ayala, A. Veidenbaum, and M. López-Vallejo, "Power-aware compilation for register file energy reduction," *International Journal of Parallel Programming*, vol. 31, no. 6, 2003.