2018 51st Annual IEEE/ACM International Symposium on Microarchitecture

FineReg: Fine-Grained Register File Management for Augmenting GPU Throughput

Yunho Oh, Myung Kuk Yoon*, William J. Song, and Won Woo Ro
School of Electrical and Electronic Engineering
Yonsei University
Seoul, Korea
{yunho.oh, myungkuk.yoon, wjhsong, wro}@yonsei.ac.kr

***Abstract*—Graphics processing units (GPUs) include a large amount of hardware resources for parallel thread executions. However, the resources are not fully utilized during runtime, and observed throughput often falls far below the peak performance. A major cause is that GPUs cannot deploy enough number of warps at runtime. The limited size of register file constrains the number of cooperative thread arrays (CTAs) as one CTA takes up a few tens of kilobytes of registers. We observe that the actual working set size of a CTA is much smaller in general, and therefore there is room for additional CTAs to run. In this paper, we propose a novel GPU architecture called *FineReg* that improves overall throughput by increasing the number of concurrent CTAs. In particular, FineReg splits the monolithic register file into two regions, one for active CTAs and another for pending CTAs. Using FineReg, the GPU begins normal executions by allocating all registers required by active CTAs. If all warps of a CTA become stalled, FineReg moves the live registers (i.e., working set) of CTA to the pending-CTA region and launches an additional CTA by assigning registers to the newly activated CTA. If the registers of either active or pending-CTA region are used up, FineReg stops introducing additional CTAs and simply performs context switching between active and pending CTAs. Thus, FineReg increases the number of concurrent CTAs by reducing the effective size of per-CTA registers. Experiment results show that FineReg achieves 32.8% of performance improvement over a conventional GPU architecture.**

***Index Terms*—GPU, Register File, Thread-Level Parallelism, Performance**

I. Introduction

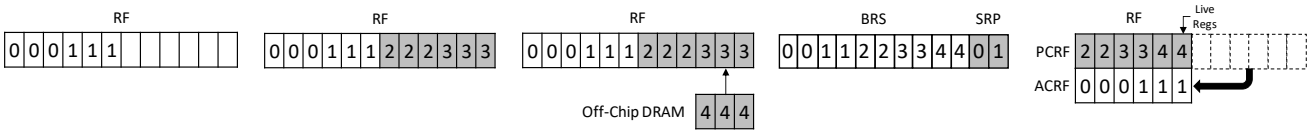
Graphics processing units (GPUs) have become the architectural choice to achieve high throughput in general-purpose computing. Thread-level parallelism (TLP) in GPUs is implemented by concurrently executing a large number of threads. Greater degree of TLP offers more chances to hide stall cycles by switching executions between threads. However, the number of concurrently executable threads in GPUs is limited by multiple factors encompassing the size of register file, shared memory, or thread scheduling resources. Prior work showed that the location of performance bottleneck varied among these hardware factors depending on workload behaviors [1], [45].

Yoon et al. showed in their work [45] that reaching the scheduling limit of a GPU still did not use a significant portion of on-chip local memory. The authors proposed Virtual Thread technique that utilized the unused portion of register file to schedule additional cooperative thread arrays (CTAs). However, the size of register file in a streaming multiprocessor (SM) puts a hard limit on increasing TLP in this case. Zorua [39] introduced CPU-like context switching in GPUs to overcome the scheduling limit imposed by the size of register file. It keeps the context information of inactive CTAs in off-device memory and copies it to the register file when the corresponding CTAs become activated. This technique allows scheduling more CTAs beyond the limit of register file size. However, storing the context of a CTA requires a non-trivial amount of space; a few tens of kilobytes as opposed to a few tens of bytes of a CPU process [42]. Zorua requires frequent data movement between the register file and off-device memory when CTA context switching occurs. Since memory operations are critical to GPU performance [16], [20], [25], [33], [35], [38], [43], increased memory traffic diminishes overall performance gain.

In this paper, we propose a novel GPU architecture named *FineReg* that improves throughput by increasing the number of concurrent CTAs via fine-grained register file management. In particular, CTAs are categorized into two groups, active and pending CTAs. Active CTAs are the ones being executed by SMs, and pending CTAs refer to the stalled ones. FineReg splits the monolithic register file into two regions, one for active CTAs and another for pending CTAs. These are referred to as *active CTA register file (ACRF)* and *pending CTA register file (PCRF)*, respectively. Executing a CTA in FineReg begins normally by allocating all required registers in the ACRF. We observe that only a fraction of ACRF holds useful values at runtime, and we refer to them as *live registers*. In most cases, the size of live registers is much smaller than the ones initially allocated in the ACRF. This gives an opportunity to schedule more CTAs by shrinking the effective register usage of individual CTAs. When all the warps of a CTA are stalled, FineReg moves only the working set (i.e., live registers) of stalled CTA into the PCRF. FineReg appends an index table to the PCRF to quickly retrieve the registers of pending CTAs. Each entry of the index table points to a location in the PCRF that stores the first live register of a pending CTA. A PCRF entry contains a tag that shows the location of next live register of the same CTA. The live registers of pending CTA are retrieved by traversing the chain of PCRF entries. As a result,

*This work was done while the author was at Yonsei University.

978-1-5386-6240-3/18/$31.00 ©2018 IEEE
DOI 10.1109/MICRO.2018.00037



(a) Baseline (b) Virtual Thread (c) Zorua (d) RegMutex (e) FineReg

Figure 1. Comparison of register file management among Baseline, Virtual Thread [45], Zorua [39], RegMutex [17], and the proposed FineReg. Virtual Thread allocates additional CTAs (CTA #2 and #3) until the register file becomes full, and Zorua places another CTA (CTA #4) in pending mode at the off-chip DRAM. RegMutex can allocate more CTAs (CTA #2-4), but only a few of them (CTA #0 and #1) can run simultaneously. FineReg splits the register file into active and pending regions (ACRF and PCRF), where the PCRF stores only the live registers of stalled CTAs under the same register file size constraint.

FineReg allows executing a greater number of CTAs beyond the defined scheduling limit. It also enables the quick initiation of pending CTAs without introducing complicated scheduling circuitry, large register renaming logic, or off-chip memory traffic to retrieve the context information.

The workflow of FineReg utilizes compile-time analysis that generates the live register information of instructions at the end of a kernel. The list of live registers is stored in global memory when the kernel launches in the GPU. CTAs begin executions in the conventional manner by allocating all the registers in the ACRF. If all the warps of an active CTA are stalled, FineReg uses the live register information generated by the compiler to selectively move registers to the PCRF. Then it uses the free space in ACRF to launch another CTA. If the PCRF does not have enough number of entries to store the live registers of stalled CTA, FineReg checks if CTAs in the PCRF are ready to resume executions and can be swapped with the stalled one.

Figure 1 illustrates comparison between the proposed FineReg and other approaches including Baseline (i.e., conventional GPU), Virtual Thread [45], Zorua [39], and RegMutex [17]. This example assumes that register file has 12 entries, and the Baseline can schedule up to two CTAs at a time. If each CTA uses three registers, Virtual Thread can allocate two more CTAs by utilizing the unused portion of register file. Similarly, Zorua makes use of unused registers to schedule two additional active CTAs, and one more CTA is placed at off-chip memory in pending mode. RegMutex [17] assigns reduced number of registers to each CTA, referred to as base register set (BRS). When a CTA requires excessive amount of registers beyond the BRS, it uses shared register pool (SRP) to assign additional registers. If there is not enough SRP, CTAs demanding additional registers must stall until SRP becomes available. In contrast, FineReg requires only two sets of CTA registers to increase the number of resident CTAs since only the live registers of pending CTAs are stored in the PCRF. As described, this paper makes the following contributions.

- *Reducing the effective size of register file usage:* We observe that only a small fraction of register file is in actual use during the runtime execution of CTAs, referred to as *live registers*. The proposed FineReg reduces the effective size of CTA register usage by storing only the live registers when CTAs are put into pending mode.

Table I. Simulation Setup

Configuration	Parameters
# of SMs	16
Clock frequency	1126MHz
SIMD width	32
Max # of warps per SM	64
Max # of threads per SM	2048
Max CTAs per SM	32
# of warp schedulers per SM	4
Warp scheduling	Greedy-then-oldest (GTO)
Register file size per SM	256KB
Shared memory size per SM	96KB
L1 cache size per SM	48KB, 8-way
L2 shared cache size	2048KB, 8-way
Off-chip DRAM bandwidth	352.5 GB/s

- *Improved utilization of scheduling hardware resources:* Fine-grained register file management of *FineReg* increases the total number of CTAs beyond the defined scheduling limit. It splits the register file into two regions, i) ACRF for the normal execution of active CTAs and ii) PCRF to store only the live registers of pending CTAs.
- *Compiler+microarchitecture optimization:* Our proposed technique utilizes the live register information generated by compilers. The list of live registers is stored in global memory when GPU kernels launch, and FineReg consults with it when switching CTAs between the ACRF and PCRF.

II. Experiment Environment

We used GPGPU-Sim [5] and configured it to simulate GTX 980-like microarchitecture [41] as described in Table I. 18 applications listed in Table II were used in our experiments. We modified a few benchmarks to generate a large number of CTAs since their default settings did not launch enough CTAs. Applications used in experiments are categorized into two groups based on their behavioral characteristics. *Type-S* workloads show that their scheduling limit is determined by the maximum number of CTAs that scheduling resources can handle; these applications still have free space in the register file or shared memory. On the other hand, *Type-R* workloads are bounded by the size of register file or shared memory before reaching the maximum number of CTAs.

Table II. Benchmark Applications

Scheduling limit	Applications (abbreviation)
CTA/warp scheduler (Type-S)	Breath-First Search (BF) [6] BiCGStab (BI) [11] Convolution Separable (CS) [32] Fluid Dynamics (FD) [11] Kmeans (KM) [6] Monte Carlo (MC) [37] Needleman-Wunsch(NW) [6] Stencil (ST) [32] Symmetric Rank 2k (SY2) [11]
Register or shared memory (Type-R)	Transpose Vector Multiply (AT) [11] CFD Solver (CF) [6] Hotspot (HS) [6] LIBOR (LI) [5] Lattice-Boltzmann (LB) [37] SGEMM (SG) [11] Sradv2 (SR) [6] Two Point Angular (TA) [37] Transpose (TR) [32]

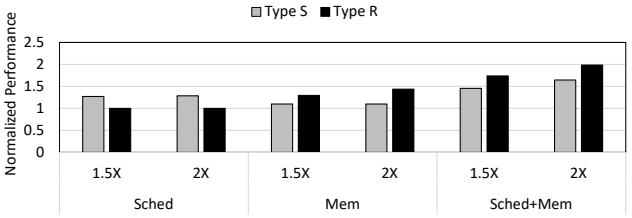


Figure 2. Performance improvements by increasing scheduling resources (Sched), on-chip memory (Mem), and both (Sched+Mem). Type-S and Type-R workloads benefit from increased scheduling resources and memory size, respectively.

III. Problems of GPU Register File

This section describes how GPUs miss opportunities to increase throughput because of register file inefficiency.

A. Limiting Factors to Increase TLP

Supporting large degree of TLP needs a tremendous amount of on-chip memory (i.e., register file, shared memory) and scheduling resources. Figure 2 shows the performance impact of scheduling resources and on-chip memory when their handling capacities are increased by 1.5x to 2x. This figure demonstrates whether contemporary GPUs are provisioned enough number of scheduling resources or memory. Results show that Type-S applications have 27.1% and 28.4% performance improvements by increasing scheduling resources by 1.5x and 2x, respectively. However, these applications have little changes when larger register file and shared memory are supplied. Type-R workloads exhibit the opposite behaviors in that they have 29.5% and 43.6% improvements for 1.5x and 2x size relaxation. If both scheduling resources and on-chip memory are sufficiently provided, Type-S and Type-R applications show 45.5% and 98.6% performance improvements.

Introducing additional scheduling resources and on-chip memory increases hardware complexity and costs. Since every

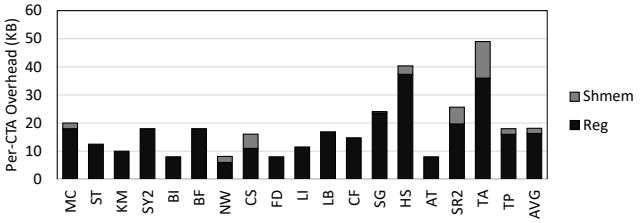
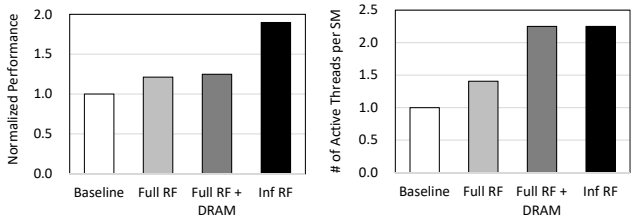


Figure 3. Overhead of shared memory (Shmem) and registers (Reg) to allocate an additional CTA. Registers account for 88.7% of total memory overhead.



(a) Normalized performance (b) Number of active threads

Figure 4. Normalized performance and active thread count of four different GPU configurations. Full RF (Virtual Thread-like setup [45]) and Full RF+DRAM (Zorua-like setup [39]) still show a large performance gap to the ideal hardware.

CTA has large register file and shared memory usage, adding more CTAs incurs significant overhead. For instance, Figure 3 shows that running an extra CTA requires 6KB to 37.3KB of overhead. Registers account for 88.7% of total memory overhead, and the rest comes from shared memory. Thus, limited on-chip memory size is a critical barrier to increase the throughput of GPUs.

B. Inefficiency of Large CTA Register File

Increasing the number of concurrent CTAs requires correspondingly complicated scheduling resources. Two previous studies, Virtual Thread [45] and Zorua [39], proposed hardware-based CTA switching methods to increase the number of CTAs over the given scheduling limit. However, both cases still leave large headroom for throughput improvement. Figure 4 shows an example with Convolution Separable (CS) benchmark [32]. The baseline execution of this application shows Type-S behaviors, where its performance is bounded by limited scheduling resources. We examined this application with three different GPU configurations. The first setup (denoted by Full RF in Figure 4) keeps launching CTAs regardless of scheduling limit (i.e., maximum number of CTAs defined by hardware specification) until the register file becomes completely full. On top of the first design, the second type (Full RF + DRAM) attempts to allocate all CTAs created by the application. Since the register file cannot hold the entire number of CTAs, the remaining ones are placed in the off-chip DRAM. The last configuration corresponds to ideal hardware with unlimited scheduling resources and memory space.

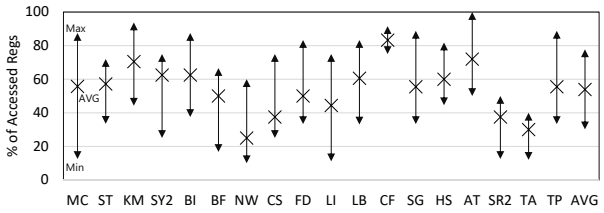


Figure 5. Percentage of register file usage in 1,000-instruction window with upper and lower bounds shown for individual benchmarks. On average, only 55.3% of register file is in actual use.

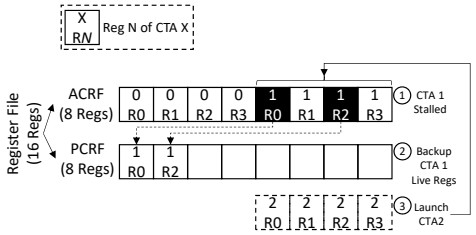
Figure 4 shows the normalized performance and number of active threads per SM for each GPU configuration. Using the first configuration achieves 21.3% performance improvement over the baseline, since it allows scheduling additional CTAs by utilizing the unused portion of register file. If the register size constraint is relaxed by placing unallocated CTAs in the off-chip memory (i.e., Full RF + DRAM), it delivers only 3.5% better performance than the first case even though twice more CTAs are concurrently scheduled. Thus, results show that there still exists a large gap to the ideal hardware.

Scheduling a CTA requires tens of kilobytes of registers, but we note that only a small fraction of them are in actual use. Figure 5 plots the average usage of register file ($\times$ marks) with upper and lower bounds for benchmarks listed in Table II. We measured the percentage of accessed registers within the statically allocated register file. On average, only 55.3% of registers were in actual use in 1,000-instruction frames. The worst case of MC, NW, LI, SR2, and TA benchmarks touched even less than 15% of registers. This observation gives an insight about the inefficiency of large CTA register file, and thus GPU throughput can be improved by storing only the in-use registers, referred to as *live registers*.

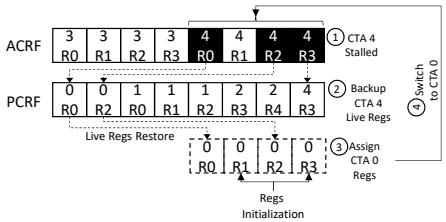
IV. Overview of FineReg Architecture

We propose a novel GPU architecture named *FineReg* to tackle the inefficiency problems of large CTA register file discussed in Section III. The objective is to maximize the number of concurrent CTAs, given the register file size, by shrinking the effective register usage of CTAs. With limited scheduling resources and memory size, not all CTAs can be concurrently scheduled. We define *active CTAs* as the ones that are being executed, and *pending CTAs* are the stalled ones that are temporarily placed in a spare memory region. Collectively, *resident CTAs* refer to active and pending CTAs altogether. The proposed FineReg assumes that runtime context switching between CTAs is supported by default.

The monolithic register file is split into two separate regions, *active CTA register file* (ACRF) and *pending CTA register file* (PCRF). The ACRF stores all the registers of active CTAs, and it is analogous to the baseline register file. The PCRF works as backup CTA register storage in which only the live registers of pending CTAs are maintained. When a pending CTA is activated (i.e., switching to active status), FineReg



(a) A case when the PCRF has free slots.



(b) A case when the PCRF is fully occupied.

Figure 6. Cases of register file management in FineReg. (a) If the PCRF is not fully occupied, FineReg initiates a new CTA (CTA 2) after moving the live registers of stalled CTA (CTA 1) to PCRF. (b) If the PCRF is full, no additional CTAs can be introduced. FineReg performs only CTA switching between the ACRF and PCRF.

restores only the live portion of register file by consulting the PCRF. The modified register file structure and support of CTA switching require an efficient scheduling scheme to trace the status of resident CTAs. If this is implemented via register file virtualization [12], 44KB of renaming table would be needed for 512 warps. Optimization using register lifetime still requires 11KB of renaming table. The proposed FineReg architecture effectively increases the efficiency of GPU register file and thus overall throughput without introducing such a large hardware overhead.

A. Workflow of FineReg Architecture

Figure 6 depicts the workflow of register file management in FineReg. At the beginning of kernel execution, an SM allocates CTAs while the ACRF is available. If all warps of an active CTA become stalled during the execution, FineReg identifies the live registers of the last instruction of each warp. If the PCRF has enough number of empty slots to accommodate the live registers as shown in Figure 6.(a), they are copied to it (R0 and R2 in the figure). Then, another CTA launches by taking over the ACRF register space that was previously occupied by the evicted CTA. In the opposite scenario in which the PCRF does not have free entries as shown in Figure 6.(b), it is no longer possible to initiate a new CTA by relocating a stalled CTA to the PCRF. FineReg in this case performs only context switching between CTAs, if it can find a pair of CTAs, one in the ACRF that is stalled (CTA 4 in the figure) and the other in PCRF that is ready to resume its execution (CTA 0). Switching CTAs between

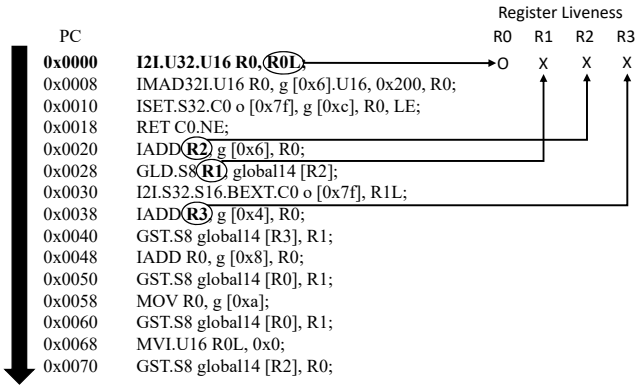


Figure 7. An example of detecting the live registers of a warp in CFD Solver [6]. A register is regarded as alive if it is used as the source operand of any following instructions until the register is used again as a destination of another instruction.

the ACRF and PCRF is allowed as long as the number of live registers of stalled CTA is smaller than or equal to the number of free slots in the PCRF including the ones that would become available by relocating the selected pending CTA to ACRF.

B. Reducing The Register Usage of Pending CTAs

Locating the live registers of stalled CTAs requires a priori information from a compiler. In particular, the compiler generates the list of live registers of every instruction and store the list in global memory for the future use by FineReg.

Figure 7 illustrates how the compiler detects the live registers of a warp. This example shows a part of CFD Solver (CS) benchmark [6] with four architectural registers (R0-R3) in use. A register is regarded as alive if it is used as the source operand of any subsequent instructions until the first encounter of an instruction that uses this register as a destination. If the execution of a warp stalls at program counter (PC) address of 0x0000 in the example code, R0 is regarded as a live register since it is used as a source operand of the first instruction. However, other registers (i.e., R2-R3) do not become live registers since they are used as destination operands at PC 0x0028, 0x0020, and 0x0038, respectively. Thus, the warp stalled at PC 0x0000 needs to keep only R0 as a live register for future executions.

Live register detection schemes were previously employed by register file virtualization [12] and Regless schemes [19]. However, the use of detection scheme in FineReg differs from prior work in that FineReg aims at increasing the number of concurrent CTAs and thus overall throughput, as opposed to reducing the size of register file for power efficiency.

C. CTA Context Management

CTA warps often stall simultaneously after a short execution period. Such a behavior offers an opportunity to increase the number of resident CTAs by employing a CTA switching mechanism. For instance, Table III shows the average number of clock cycles until CTAs become completely stalled. We measured the duration between the first instruction issue of any warps and complete stall of all warps. The average duration varied across applications, but CTAs became completely stalled within 300-2,300 clock cycles. The observation proves the necessity of CTA switching mechanism to hide such frequent CTA stalls.

Table III. Average CTA Execution Time Until Complete Stall

App.	# cycles	App.	# cycles	App.	# cycles
MC	1525	ST	1503	KM	892
SY2	1245	BI	1338	BF	193
NW	311	CS	512	FD	2018
LI	1021	LB	828	CF	955
SG	2299	HS	752	AT	1272
SR2	774	TA	1054	TP	775

Scheduling CTAs over the scheduling limit requires a cost-effective CTA switching mechanism. FineReg implements a CTA status monitoring module to keep the pipeline context and register status of all resident CTAs. The pipeline context of a CTA indicates whether all warps of the CTA are being executed through the pipeline or backed up in the shared memory. This information is used to determine which CTAs can be moved from the execution pipeline to shared memory, and vice versa. Similarly, register status shows if the CTA registers are present in the ACRF or PCRF. Using this information, FineReg can make decisions if the registers of the CTA should be restored from PCRF to ACRF or backed up in the opposite way. Detailed microarchitectural implementations are described in Section V.

V. FineReg Implementation

Figure 8 shows the GPU microarchitecture using FineReg. In contrast to the baseline GPU design, the register file is split into active and pending-CTA regions, as drawn in the read-operands stage. The ACRF works in the same way as the baseline register file, and PCRF stores the reduced register set (i.e., live registers) of pending CTAs. CTA status monitor maintains the information of resident CTAs. Register management unit (RMU) is annexed to the register file structures to keep the live register information of active CTAs and register indices of pending CTAs. FineReg refers to the RMU when it needs to switch CTAs between the ACRF and PCRF. In addition, FineReg implements CTA switching logic that we adopted from that of Virtual Thread [45].

A. Compiler Support

FineReg requires compiler support to generate the list of live registers per instruction. We adopted similar detection techniques used in register file virtualization [12], Regless [19], and cache-emulated register file [15]. For each static instruction, the compiler analyzes all the registers used by its subsequent instructions. If a register is used as the source operand of any following instructions, the register is regarded as alive until it is used again as a destination by another instruction. This process is iterated for every instruction to compose the list of live registers. The list is saved as a simple bit vector that marks the liveness of each register at

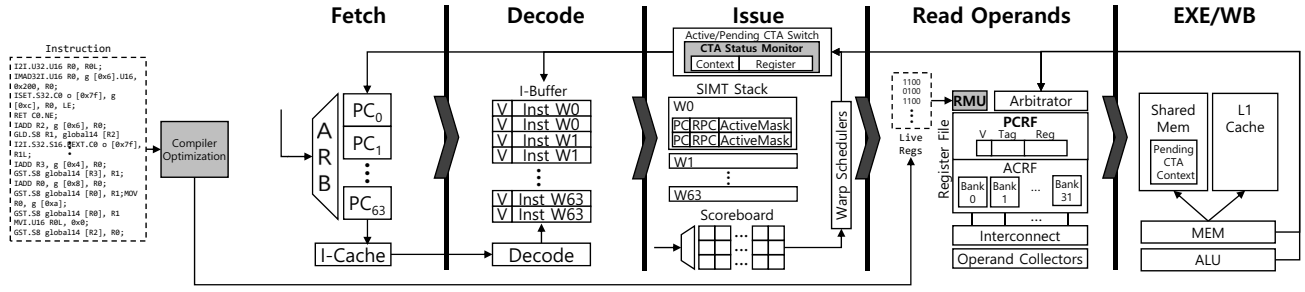
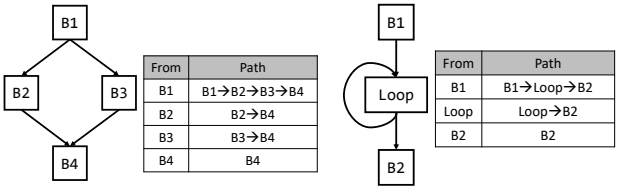


Figure 8. FineReg microarchitecture. Issue stage includes CTA status monitor that maintains the context and register information of resident CTAs. Register file consists of ACRF and PCRF that handles active and pending CTAs, respectively. Register management unit (RMU) is annexed to the read-operands stage to control register transfers between ACRF and PCRF.



(a) Diverging branch (b) Loop

Figure 9. Detecting live registers in a diverging branch or loop. (a) Compiler needs to traverse all basic blocks to collect live register information. (b) Loop requires visiting each block only once to compose live register list.

the corresponding position. The bit vector is 64-bit long (i.e., maximum number of registers per thread), and it is initially stored in a reserved area at the off-device memory.

If a kernel includes diverging branches, the compiler traverses the static instructions of basic blocks along the control paths. Using a post-dominator (PDOM) branch divergence control method, a warp has to sequentially visit diverging branches to make its threads execute all basic blocks [7], [8], [36]. Figure 9.(a) shows an example of diverging branch with four basic blocks. B2 and B3 branch off from B1, and B4 is a re-convergence point. When the compiler analyzes the static instruction of B1, it has to traverse all basic blocks in the sequence of B1, B2, B3, and B4. However, collecting live register information at B2 (or B3) needs to visit only B2 and B4 (or B3 and B4). Compiler traversal in a loop requires visiting each basic block only once, as shown in Figure 9.(b). For instance, analyzing a warp stalled at B1 visits a single loop body and B2.

B. CTA Switching and Status Monitoring

FineReg requires a CTA switching mechanism to support scheduling CTAs beyond the baseline scheduling limit. A CTA switching event occurs when the execution of an active CTA is stalled. However, FineReg takes different actions depending on the state of PCRF. If the PCRF is full at the moment of CTA stall, FineReg finds if any of the pending CTAs had become ready to resume executions. If so, the live registers of stalled CTA are relocated to the PCRF, and those of a selected pending CTA are restored to the ACRF. In case that the PCRF is readily available, another CTA is initiated after moving the live registers of stalled CTA to the PCRF.

Table IV. Encoding of CTA Context and Register Status

Status types	0	1	2
Context location	CTA Not launched	Shared memory	Pipeline
Register location	CTA Not launched	PCRF	ACRF

A CTA status monitor maintains the information of CTA contexts and registers, as shown in the issue stage of Figure 8. Each field of the status monitor is implemented as an array whose elements are 2-bit values. An element value indicates the location of context and registers of corresponding CTA. Table IV summarizes the encoding of status information stored at the arrays. A CTA is regarded as active only if both context and register fields are set to 2 (i.e., `0b10`). Otherwise, the CTA is in pending mode. The CTA status monitor serves to quickly prioritize pending CTAs when FineReg needs to make switching decisions. For instance, FineReg first attempts to select a CTA whose context field is set to 1 and register field is 2. If no such CTAs are found, others with both context and register fields set to 1 become the next candidates. If conditions are satisfied, FineReg performs CTA switching, followed by updating the fields of CTA status monitor.

In rare situations, the live registers of stalled CTA may fail to relocate to the PCRF when it does not have enough number of free entries even if one of the pending CTAs is swapped out. The stalled CTA then has to remain in the ACRF until there are enough vacancies in the PCRF or its execution resumes.

C. Register Management Unit

Register management unit is one of the key constituents of FineReg. It is comprised of five components; i) live register information cache, ii) register index decoder, iii) PCRF pointer table, iv) free space monitor, and v) PCRF access logic. The structure of RMU is drawn in Figure 10. If FineReg decides to switch CTAs, it looks up the PC of warps of a stalled CTA. The RMU retrieves the bit vector (i.e., live register list) mapped to

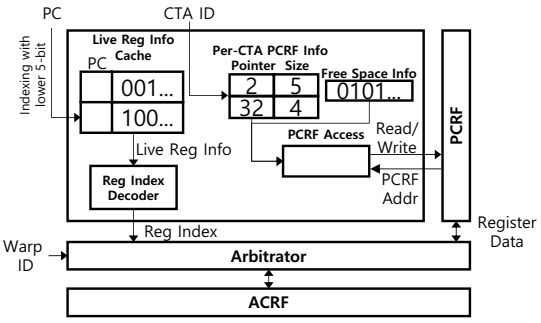


Figure 10. Register management unit is comprised of five components, i) live register information cache, ii) register index decoder, iii) PCRF pointer table, iv) free space monitor, and v) PCRF access logic.

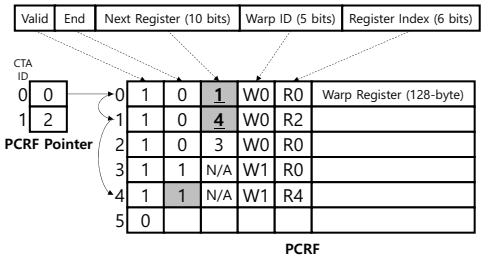


Figure 11. PCRF includes tag (valid, end, next register, warp ID, register index) and 128-byte register fields. PCRF pointer directs to the first live register of a CTA, and the next register pointer in the tag shows the location of subsequent live registers of the same CTA.

the PC. This process incurs long latency since the bit vector resides in the off-chip memory.

It is observed that a few specific types of instructions stall warp executions, but the culprit instruction types vary depending on workload characteristics. Since all the warps execute the same instruction stream, only a small number of instructions in a kernel are responsible for causing stalls [4], [9], [20], [22], [31], [33], [38], [44]. This observation leads to devising a cache to store the bit vector of live registers in the RMU. The bit vector cache is implemented as a direct-mapped structure, and each block is 64 bits long. We have empirically obtained that 32 entries are sufficient for the cache to effectively serve FineReg. The cache is indexed by hashing 5 bits of PC address, and the PC tag is compared with the current PC (see Figure 10). If a matching entry is found, the cache transfers the bit vector to the register index decoder. Register indices are generated by the decoder and sent to the ACRF through arbitrator. If there is a pending CTA that had become ready to resume executions, registers are exchanged between the ACRF and PCRF.

The RMU includes PCRF pointer table, where each entry contains the count and indices of live registers of a pending CTA. Table entries are differentiated by using CTA IDs. The free space monitor is used in the RMU to provide the number of empty slots and their location in the PCRF. The free space information is maintained via an array whose elements are simple 1/0 flags; 0 indicates that the corresponding PCRF entry is empty, and 1 means the register space is occupied.

D. Register File Design

The register file design of FineReg is composed of ACRF and PCRF. The ACRF and PCRF combined together have the same register file size of the baseline GPU. The ACRF is implemented and works the same as the baseline register file, and the PCRF serves as a backup register space for stalled CTAs. Figure 11 depicts the tag structure and operations of PCRF. FineReg accesses the PCRF by using indices obtained from the RMU. A PCRF entry is comprised of a tag and 128-byte registers. The tag includes valid and end bits, next register pointer, warp ID, and register index. The end bit is used to mark if the corresponding register entry is the last one in the live register list of a CTA. The 10-bit pointer is used to show the location of the next live register of the CTA. 5 bits are used as a warp ID since a CTA can have up to 32 warps, and a register index has 6 bits to cover 63 per-thread registers.

E. Register File Operations

FineReg performs CTA switching using the following procedure. It first determines if one of the pending CTA is ready for execution and thus can be moved to the ACRF. If found, the RMU calculates the number of free entries in the PCRF by adding the count of readily empty slots to the ones that would become available if the selected pending CTA moves out. If no pending CTA can move back to the ACRF, the number of free entries is simply that of unoccupied ones. Finding the number of free slots can be done by aggregating the number of 0 bits (i.e., empty status) in the free space monitor array. FineReg performs CTA switching if the number of free entries in the PCRF is greater than or equal to the live register count of a stalled CTA that needs to be evicted from the ACRF.

The RMU restores the live registers of pending CTA through arbitrator, if necessary, starting from the entry (i.e., first live register) directed by the PCRF pointer table. Meanwhile, the live registers of stalled CTA in the ACRF are transferred to the PCRF. Reading ACRF registers takes multiple cycles to complete, so a 128-byte buffer is used to keep the registers read from the ACRF. The next register tag of newly placed PCRF entries is set for the future restoration procedure. Then, the RMU updates the free space monitor to reflect changes in the PCRF.

Figure 11 illustrates an example of PCRF access operations. The example assumes that a CTA using the ACRF has stalled and needs to be swapped with a pending CTA (CTA 0 in the example) backed up in the PCRF. The RMU first reads the entry point of live register list, indicated by the PCRF pointer. For instance, the PCRF pointer of CTA 0 directs to the top entry (0th index) of PCRF. This register is thus read and restored in the ACRF. The tag of 0th PCRF entry indicates that the next live register is located at index 1, and this entry again shows that the 4th entry has the subsequent

live register. As described, FineReg traverses the chain of live registers to retrieve the complete register list of the CTA. At the encounter of end bit (set at the 4th entry in the example), it stops searching live registers. Finally, the live registers of stalled CTA are placed in the PCRF.

It takes at least four clock cycles to access a PCRF tag and the corresponding register. Retrieving the chain of live registers is pipelined, but it may take several hundreds of clock cycles depending on the size of live registers of a pending CTA. However, CTA switching latency can be effectively hidden by executing other active warps, and thus performance penalty due to CTA switching is hardly observed in FineReg.

F. Hardware Overhead

FineReg is designed to support up to 128 CTAs or 512 warps. The CTA status monitor then requires 256 bits (= 2 bits/CTA $\times$ 128 CTAs) for each context and register field. The live register bit vector cache has 12-byte entries; 4 bytes are for PC, and 64 bits are used as a bit vector. Since the cache has 32 entries, it takes total 384 bytes. The PCRF information table has 128 lines, and each line is composed of 10-bit PCRF pointer and 6-bit live register count. Thus, it requires total 256 bytes of SRAM. Lastly, the PCRF has 2.15KB of tag overhead, for 21 bits per tag times 1,024 registers. The data field of PCRF is not counted as hardware overhead since FineReg does not use more registers with ACRF and PCRF combined, compared to the register file size of baseline GPU architecture. Instead, the CTA switching logic of FineReg requires 2.4KB of storage overhead [45]. In sum, FineReg implementation requires about 5.02KB of additional storage, which is translated to only 0.38% of SM area overhead in Fermi GPU [23], [41], based on an estimation using CACTI [30].

Similarly, storing the 12-byte bit vector of instructions consumes a tiny amount of off-chip memory space. In particular, each application used in our experiments had only up to 600 static instructions. It means that 4.8KB of off-chip memory space would be sufficient to store bit vectors per application. Since only a small number of instructions cause long-latency stalls, FineReg accesses only a few bit vectors at off-chip memory, which is not a significant performance penalty.

VI. FineReg Evaluation

A. Evaluation Methodology

We used GPGPU-Sim [5] for our analysis, and the baseline model was set up based on the configuration in Table I. We updated the simulation framework to reflect FineReg microarchitecture, and experiments were made with 18 benchmarks listed in Table II. The size of PCRF in FineReg was set to 128KB, i.e., a half of baseline register file size. The performance of FineReg was compared against Virtual Thread [45], Reg+DRAM mode that implemented Zorua-like architecture [39], and RegMutex [17]. In particular, Virtual Thread enables executing a large number of CTAs over the baseline scheduling limit by initiating CTAs until register file or shared memory becomes full. In contrast to FineReg, Virtual Thread simply allocates all the registers of CTAs regardless of their liveness

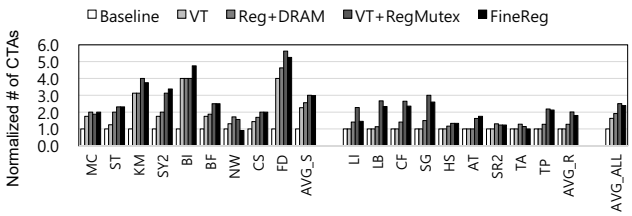


Figure 12. Number of concurrent CTAs in four different GPU configurations. FineReg schedules 111.8% more CTAs over the baseline on average, and it also outperforms Virtual Thread (VT) and Reg+DRAM by 48.6% and 20.1%, respectively. VT+RegMutex schedules 11.5% more CTAs than FineReg.

at runtime. Reg+DRAM adds CTA switching feature to Virtual Thread architecture. It also keeps launching CTAs until register file becomes full but is capable of allocating additional CTAs by utilizing a portion of off-chip DRAM. When the execution of a CTA is stalled, the scheduler performs CTA switching with the one located in the off-chip DRAM if it is ready to run. This implementation is similar to Zorua, except that we varied the number of pending CTAs in the off-chip memory to find its best-performance setup for every application. We merged Virtual Thread into RegMutex to empirically find the optimal operating point of RegMutex (i.e., the ratio of BRS and SRP) since we learned that the BRS/SRP ratio used in the work of Khorasani et al., [17] did not necessarily produce the peak performance of RegMutex.

B. Impact on Thread-Level Parallelism

Figure 12 shows the number of concurrent CTAs for individual benchmarks. Results show that FineReg is capable of running 141.7% more CTAs than the baseline, which is again 48.6% and 20.1% greater than Virtual Thread and Reg+DRAM, respectively. Differences are due to the efficiency of register file utilization in that FineReg shrinks the effective size of register file usage. In particular, FineReg increased the number of CTAs by 203.8% for Type-S workloads but only 79.8% for Type-R applications. The performance of Type-S applications is bounded by scheduling resources instead of register file of shared memory size. It indicates that they have relatively small register file footprints (or live registers), and thus compacted register usage significantly increases the number of concurrent CTAs. On the contrary, increase in concurrent CTA count is limited for Type-R benchmarks since they have relatively large register file usage. Such a workload characteristic leaves smaller room for additional CTAs to join.

Similarly, Virtual Thread increases CTA count by 126.3% for Type-S applications, but it shows no improvement over the baseline for Type-R workloads. Reg+DRAM performs better than the Virtual Thread in that it increases CTA count by 155.9% and 27.7%, respectively. However, its performance improvement is minimal since CTA context switching incurs long latencies to access off-chip DRAM. VT+RegMutex initiates 11.5% more CTAs than FineReg on average, and especially 21.5% more CTAs for Type-R applications. It is due to reduced register set size that helps cram as many CTAs as possible.

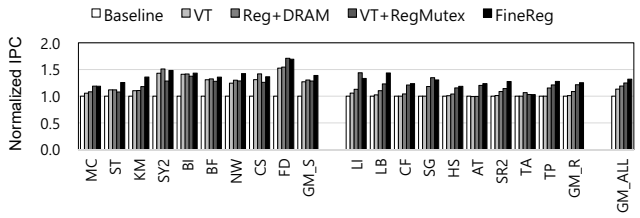


Figure 13. Normalized instructions-per-cycle of four GPU configurations. FineReg provides 32.8% performance improvement over the baseline on average. Importantly, increases in the number of concurrent CTAs are not directly translated to performance enhancement.

C. Performance Impact

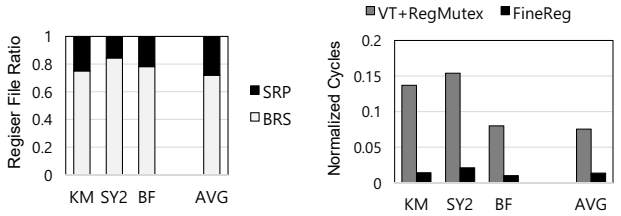
The performance impact of four GPU configurations is plotted in Figure 13. It shows that FineReg delivers 32.8% improvement over the baseline. The improvement is attributed to enhanced TLP, i.e., number of concurrent CTAs. FineReg also outperforms Virtual Thread, Reg+DRAM, and VT+RegMutex by producing 18.5%, 12.8%, and 7.1% greater throughput.

However, the increased number of concurrent CTAs does not always directly translate to performance improvement. For example, 2x increase in the number of CTAs achieves more than 60% of performance enhancement for BI, FD, NW, and SY2 benchmarks, but 2.5x more concurrent CTAs of BF and KM benchmarks result in less than 40% of performance gain. The latter applications are highly memory-bound [18], [21], [35], where the performance gain is diminished by long-latency memory operations even though FineReg hides stall cycles by enhancing the TLP. For the same reason, Reg+DRAM provides only 2.2% better performance than Virtual Thread while running 32% more CTAs. VT+RegMutex can execute more CTAs than FineReg. However, it suffers from frequent pipeline stalls due to SRP contentions between warps, especially for memory-intensive applications such as KM, SY2, and BF. When the execution of a warp is stalled by long-latency memory instructions, it continues to occupy SRP and hinders other warps from scheduling even though they are executable. Thus, such long-latency memory instructions quickly depletes SRP, thereby creating SRP contentions between warps.

For Type-R applications, FineReg provides 20% performance improvement compared to the baseline and also outperforms Virtual Thread, Reg+DRAM, and RegMutex by 20.0%, 9.5%, and 2.1%, respectively. An exception happens with TA benchmark. This application depletes shared memory such that almost no performance gain is achievable by any configurations.

D. Stalls Caused by Register File Depletion

Both FineReg and VT+RegMutex can cause stalls depending on the utilization of PCRF and SRP. FineReg causes pipeline stalls when the PCRF has no more free space to locate the live registers of stalled CTAs. Similarly, stalls occur in RegMutex when SRP is used up, and no more



(a) SRP/BRS ratio in RegMutex. (b) Stalls due to register file depletion.

Figure 14. (a) SRP/BRS ratio in RegMutex, and (b) stalls due to the lack of SRP (RegMutex) and PCRF (FineReg). Reduced size of SRP in RegMutex causes stalls even if SMs have schedulable warps.

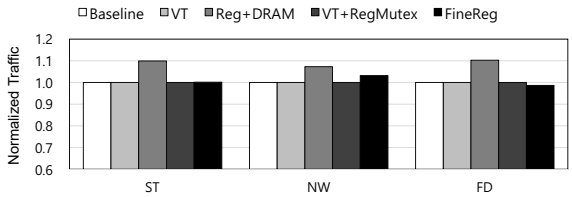


Figure 15. Comparison of off-chip memory traffic among four GPU configurations. Reg+DRAM increases overall memory traffic up to 9.9%, whereas Virtual Thread, RegMutex, and FineReg generate less than 1% additional memory traffic.

warps demanding additional registers beyond BRS can be scheduled. Figure 14 shows how many stalls are due to the unavailability of SRP (RegMutex) and PCRF (FineReg) when they have schedulable CTAs. The experiment was conducted with memory-intensive applications mentioned in Section VI-C, i.e., KM, SY2, and BF. Results show that the optimal settings of VT+RegMutex designate 28.1% of register file as SRP on average and 20.8% for the memory-intensive applications. However, reduced SRP/BRS ratio causes frequent resource contentions between CTAs, since RegMutex does not release SRP when CTAs are stalled by long-latency memory instructions. As a result, VT+RegMutex stalls applications for 7.5% of execution time, while FineReg shows only 1.3% of stall cycles due to the depletion of register file.

E. Memory Traffic Effects

We measured the traffic between off-chip memory and register file to analyze the effect of long-latency memory operations on performance. Figure 15 shows the normalized memory traffic of four GPU configurations. Three applications, FD, NW, ad ST, were specifically measured since Reg+DRAM does not show performance improvement while deploying more CTAs than Virtual Thread, RegMutex, or FineReg.

The selected benchmarks suffer from numerous L1 cache misses, and thus all GPU configurations make frequent CTA switching. Since Reg+DRAM requires transferring CTA context from register file to off-chip memory and vice versa, it generates 7.2%-9.9% more memory traffic than the baseline for the selected benchmarks. In contrast, the amount

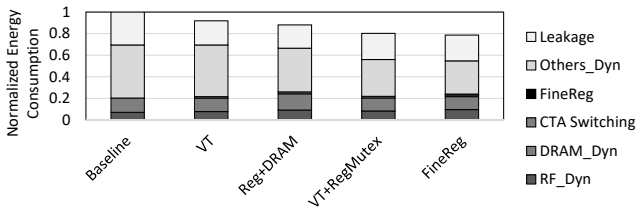


Figure 16. Normalized energy consumption of various GPU configurations. DRAM_Dyn, RF_Dyn, and Others_Dyn denote dynamic energy dissipation by off-chip DRAM, register file, and remaining logic, respectively. FineReg and CTA Switching refer to energy consumption by FingReg scheduling resources and CTA switching logic. On average, FineReg uses 21.3% less energy than the baseline.

of memory traffic differs only by 0.1% for Virtual Thread. FineReg produces 1% additional memory data compared to the baseline, where the increase is caused by transferring live register bit vectors. Such differences indicate that increased memory traffic of Reg+DRAM scheme is due to CTA context switching. Thus, the performance benefit of Reg+DRAM is significantly deteriorated by off-chip memory traffic.

F. Energy Consequences

We adopted power models used in register file virtualization [12] and GPUWattch [24] to estimate the energy consequences of various GPU configurations use in our experiments. Figure 16 shows the average energy dissipation of different configurations for all benchmarks with detailed breakdown. For most applications, performance improvements by different GPU configurations turn into energy reduction. FineReg reduces overall energy consumption by 21.3% compared with the baseline, and it also consumes 12.3%, 8.6%, and 1.5% less energy than Virtual Thread, Reg+DRAM, and VT+RegMutex, respectively. In particular, the register file of FineReg has increased energy consumption to support CTA switching, but the increase is shadowed by energy savings in renaming logic, leakage, DRAM (compared to Reg+DRAM).

G. Sensitivity Analysis

1) ACRF and PCRF size: The size of register file in FineReg affects the number of concurrent CTAs. Smaller ACRF size allows less number of active CTAs to run, which in turn increases stall cycles since there are not enough number of active CTAs to hide stalls via CTA switching. On the other hand, small PCRF decreases the chance to introduce additional CTAs as the register file quickly fills up.

Figure 17 shows the total number of concurrent CTAs with various combinations of ACRF and PCRF sizes. The size of ACRF and PCRF is varied between 64KB and 192KB while keeping the same total size. Results show that FineReg achieves the best performance when the size of both ACRF and PCRF are equally balanced at 128KB each. At this configuration, FineReg runs 2.47x more CTAs compared to the baseline, among which active CTAs account for only 33%. The combination of 160KB ACRF and 96KB PCRF

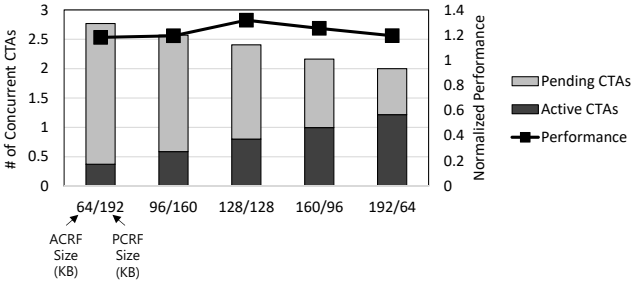
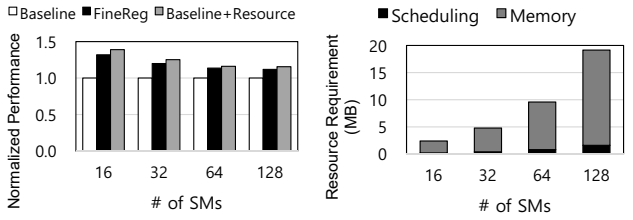


Figure 17. Performance sensitivity with respect to register file size. The peak performance is achieved when the size of both register files are balanced. The highest degree of TLP (64KB/192KB case) or equal number of active and pending CTAs (160KB/96KB case) does not necessarily produce the best performance.



(a) Performance (b) Baseline+Resource overhead

Figure 18. (a) Normalized performance of FineReg and Baseline+Resource with varying number of SMs, and (b) hardware overhead of Baseline+Resource. Baseline+Resource is a size-scaled baseline architecture that can run the same number of CTAs as FineReg for each SM configuration.

generates about the same ratio of active and pending CTAs, but it does not necessarily produce the peak performance. In fact, decreased TLP due to smaller PCRF gives 5.4% less performance than the best case. The number of total CTAs is maximized with the combination of 64KB ACRF and 192KB PCRF, where the majority of CTAs are in pending mode. Although this combination gives the highest degree of TLP, small ACRF results in frequent CTA switching and thus 12.9% degraded performance.

2) Number of Streaming Multiprocessors: Recent GPUs keep increasing the number of SMs every generation [29]. Future generation GPUs will require even greater degree of TLP with large hardware overhead. Figure 18.(a) shows the performance implication of FineReg with varying number of SMs. With increasing number of SMs from 16 to 128, FineReg consistently achieves more than 10% greater performance over the baseline architecture. For each configuration, we manually modified the baseline architecture to run the same number of CTAs as FineReg and calculated necessary hardware overhead to realize such TLP in the baseline model. Since the baseline hardware does not exploit CTA switching techniques, it demands significantly larger memory and scheduling resources to accommodate the same number of CTAs as FineReg. This

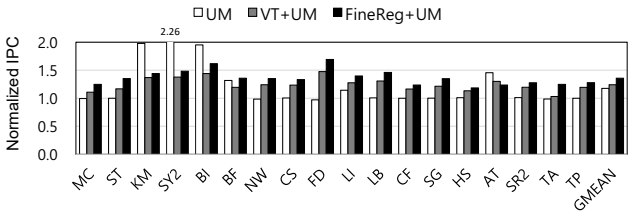


Figure 19. Normalized performance of unified on-chip local memory (UM) combined with Virtual Thread (VT) and FineReg. FineReg+UM provides the most performance enhancement, compared with other configurations.

setup is denoted as Baseline+Resource in the figure, and the corresponding hardware overhead is drawn in Figure 18.(b). The figures show that Baseline+Resource can achieve another 3.6%-5.3% performance improvement over FineReg, but it requires 2.4MB to 19.1MB of overhead, which is impractical to implement. In contrast, FineReg can produce comparable performance under a few tens of kilobytes of SRAM overhead.

3) Unified On-Chip Local Memory Structure: Unified on-chip local memory (UM) architecture [10] uses a combined storage for registers, shared memory, and data cache. We adopted this architecture to FineReg such that PCRF, shared memory, and L1 cache are coalesced into UM. We set the size of UM as 272KB (=128+96+48) in our evaluation. This implementation increases the number of possible pending CTAs as the PCRF may grow. In addition, the structure can indulge in large L1 cache if a kernel uses small number of registers and shared memory. Figure 19 shows the performance of UM, virtual thread (VT)+UM, and FineReg+UM. Several benchmarks including AT, BI, KM, and SY2 exhibit significant performance enhancement using the UM as effective L1 cache size increases. However, most of other benchmarks do not benefit much from the UM-only configuration due to limited TLP. In overall, the UM-only configuration increases performance by 17.6%. The VT+UM combination offers additional 6.7% performance improvement over the UM-only design, but aforementioned benchmarks (e.g., AT, BI, KM, SY2) no longer show significant performance increases. FineReg+UM shows similar trends as VT+UM, but it provides much better performance enhancement in overall by increasing the number of concurrent CTAs. On average, FineReg+UM shows 35.6% performance improvement, compared to the UM-only configuration. Thus, it proves that FineReg can easily be integrated with other register file structures for further enhancement.

VII. Related Work

FineReg aims at improving GPU throughput by newly designing the register file structure and its management. The performance improvement of FineReg is achieved by enhancing the utilization of large CTA register file in the GPU. The following summaries related work.

A. GPU Register File

Register files is one of the key constituents of GPU architectures to implement high-throughput computing. A GPU needs to hold the registers of thousands of active threads to provide quick data references. However, the gigantic size of register file prohibits the designs and operations of GPUs in various aspects. To increase the utilization of large GPU register file, Gebhart et al. proposed two-level warp scheduling [9] in which a small register cache is used to store the registers of active threads, and those of pending threads are maintained in a separate register file. Kloosterman et al. proposed Regless architecture [19], and Jeon et al. suggested register file virtualization [12] utilizing the liveness of GPU registers. Abdel-Majeed et al. introduced a tri-modal register access control [1], [2] and register partitioning scheme [3]. In addition, new register file architectures based on eDRAM or non-volatile memory devices were proposed to tackle power problems [13], [14], [26], [28], [40], [46].

B. TLP Enhancement

Yoon et al. suggested a hardware-based TLP enhancement called Virtual Thread [45], which enables launching CTAs over the predefined scheduling limit via swapping contexts between active and pending CTAs. Vijaykumar et al. proposed a TLP improvement method using compiler assists and hardware supports, named Zorua [39]. The proposed technique identifies the optimal number of threads per CTA in a GPU kernel. Khorasani et al. [17] introduced RegMutex that reduces the effective register set size. It divides the register file into two sections, i) BRS that contains the reduced set of registers and ii) SRP that is shared among different warps. RegMutex helps increase the number of scheduled CTAs and thus improves overall throughput by shrinking register size per warp.

C. On-Chip Local Memory

Gebhart et al. introduced the concept of unified on-chip local memory [10] that forms a single memory space that combines register file, shared memory, and cache. Jing et al. proposed a new cache-emulated register file that coalesces cache lines and register data [15]. In addition, other related work suggested cache structures that can improve GPU performance by offering fine-controlled cache lines [27], [34].

Departing from the prior work, FineReg improves GPU throughput by combining a new register file management technique with CTA switching scheme. The proposed technique does not suffer from register file contentions observed in RegMutex.

VIII. Conclusion

The proposed FingReg architecture improves GPU throughput by enhancing the utilization of large register file. FineReg delivers an insight that only a small fraction of register file is in actual use during the runtime execution of CTAs. Fine-grained register file management helps increase the total number of CTAs beyond the defined scheduling limit. FineReg provides a cost-efficient solution to implement CTA switching, and experiment results prove that it outperforms other GPU configurations proposed in the prior work [17], [39], [45].

ACKNOWLEDGEMENTS

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP) (No. NRF-2018R1A2A2A05018941), and by the Technology Innovation Program (No. 10080674, Development of Reconfigurable Artificial Neural Network Accelerator and Instruction Set Architecture) funded by the Ministry of Trade, Industry & Energy (MOTIE, Korea) and Korea Semiconductor Research Consortium (KSRC) support program for the development of the future semiconductor device. Won Woo Ro and William J. Song are the co-corresponding authors.

REFERENCES

[1] M. Abdel-Majeed and M. Annavaram, "Warped register file: a power-efficient register file for GPGPUs," *IEEE International Symposium on High Performance Computer Architecture*, Feb. 2013.

[2] M. Abdel-Majeed, D. Wong, and M. Annavaram, "Warped gates: gating-aware scheduling and power gating for GPGPUs," *IEEE/ACM International Symposium on Microarchitecture*, Dec. 2013.

[3] M. Abdel-Majeed, A. Shafaei, H. Jeon, M. Pedram, and M. Annavaram, "Pilot register file: energy-efficient partitioned register file for GPUs," *IEEE International Symposium on High Performance Computer Architecture*, Feb. 2017.

[4] N. Agarwal, D. Nellans, M. Stephenson, M. O'Connor, and S. Keckler, "Page placement strategies for GPUs within heterogeneous memory systems," *International Conference on Architectural Support for Programming Languages and Operating Systems*, Mar. 2015.

[5] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," *IEEE International Symposium on Performance Analysis of Systems and Software*, Apr. 2009.

[6] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," *IEEE International Symposium on Workload Characterization*, Oct. 2009.

[7] W. Fung, I. Sham, G. Yuan, and T. Aamodt, "Dynamic warp formation and scheduling for efficient GPU control flow," *IEEE/ACM International Symposium on Microarchitecture*, Dec. 2007.

[8] W. Fung and T. M. Aamodt, "Thread block compaction for efficient SIMT control flow," *IEEE International Symposium on High Performance Computer Architecture*, Feb. 2011.

[9] M. Gebhart, D. Johnson, D. Tarjan, S. Keckler, W. Dally, E. Lindholm, and K. Skadron, "Energy-efficient mechanisms for managing thread context in throughput processors," *ACM/IEEE International Symposium on Computer Architecture*, June 2011.

[10] M. Gebhart, S. Keckler, B. Khailany, R. Krashinsky, and W. Dally, "Unifying primary cache, scratch, and register file memories in a throughput processor," *IEEE/ACM International Symposium on Microarchitecture*, Dec. 2012.

[11] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a high-level language targeted to GPU codes," *Innovative Parallel Computing*, May, 2012.

[12] H. Jeon, G. Ravi, N. Kim, and M. Annavaram, "GPU register file virtualization," *IEEE/ACM International Symposium on Microarchitecture*, Dec. 2015.

[13] D. Jeong, Y. Oh, J. Lee, and Y. Park, "An eDRAM-based approximate register file for GPUs," *IEEE Design Test*, vol. 33, no. 1, Nov. 2015.

[14] 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," *ACM/IEEE International Symposium on Computer Architecture*, June 2013.

[15] 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," *IEEE/ACM International Symposium on Microarchitecture*, Oct. 2016.

[16] O. Kayiran, A. Jog, M. Kandemir, and C. Das, "Neither more nor less: optimizing thread-level parallelism for GPGPUs," *International Conference on Parallel Architectures and Compilation Techniques*, Sep. 2013.

[17] F. Khorasani, H. Esfeden, A. Farmahini-Farahani, N. Jayasena, and V. Sarkar, "RegMutex: Inter-Warp GPU Register Time-Sharing," *ACM/IEEE International Symposium on Computer Architecture*, June 2018.

[18] K. Kim, S. Lee, M. Yoon, G. Koo, W. Ro, and M. Annavaram, "Warped-preexecution: A GPU pre-execution approach for improving latency hiding," *IEEE International Symposium on High Performance Computer Architecture*, Mar. 2016.

[19] J. Kloosterman, J. Beaumont, D. Jamshidi, J. Bailey, T. Mudge, and S. Mahlke, "Regless: just-in-time operand staging for GPUs," *IEEE/ACM International Symposium on Microarchitecture*, Oct. 2017.

[20] G. Koo, Y. Oh, W. Ro, and M. Annavaram, "Access pattern-aware cache management for improving data utilization in GPU," *ACM/IEEE International Symposium on Computer Architecture*, June 2017.

[21] N. Lakshminarayana and H. Kim, "Spare register-aware prefetching for graph algorithms on GPUs," *IEEE International Symposium on High Performance Computer Architecture*, Feb. 2014.

[22] J. Lee, D. Woo, H. Kim, and M. Azimi, "Green cache: exploiting the disciplined memory model of OpenCL on GPUs," *IEEE Transactions on Computers*, vol. 64, no. 11, Jan. 2015.

[23] S. Lee, K. Kim, G. Koo, H. Jeon, W. Ro, and M. Annavaram, "Warped-compression: enabling power-efficient GPUs through register compression," *ACM/IEEE International Symposium on Computer Architecture*, June 2015.

[24] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. Kim, T. Aamodt, and V. Reddi, "GPUWattch: enabling energy optimizations in GPGPUs," *ACM/IEEE International Symposium on Computer Architecture*, June 2013.

[25] D. Li, M. Rhu, D. R. Johnson, M. O'Connor, M. Erez, D. Burger, D. Fussell, and S. Keckler, "Priority-based cache allocation in throughput processors," *IEEE International Symposium on High Performance Computer Architecture*, Feb. 2015.

[26] G. Li, X. Chen, G. Sun, H. Hoffmann, Y. Liu, Y. Wang, and H. Yang, "A STT-RAM-based low-power hybrid register file for GPGPUs," *ACM/EDAC/IEEE Design Automation Conference*, June 2015.

[27] L. Li, A. Hayes, S. Song, and E. Zhang, "Tag-split cache for efficient GPGPU cache utilization," *International Conference on Supercomputing*, June 2016.

[28] X. Liu, M. Mao, X. Bi, H. Li, and Y. Chen, "An efficient STT-RAM-based register file in GPU architectures," *Asia and South Pacific Design Automation Conference*, Jan. 2015.

[29] U. Milic, O. Villa, E. Bolotin, A. Arunkumar, E. Ebrahimi, A. Jaleel, A. Ramirez, and D. Nellans, "Beyond the socket: NUMA-aware GPUs," *IEEE/ACM International Symposium on Microarchitecture*, Oct. 2017.

[30] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "CACTI 6.0: a tool to model large caches," *HP Laboratories Technical Report*, HPL-2009-85, Apr. 2009.

[31] V. Narasiman, M. Shebanow, C. Lee, R. Miftakhutdinov, O. Mutlu, and Y. Patt, "Improving GPU performance via large warps and two-level warp scheduling," *IEEE/ACM International Symposium on Microarchitecture*, Dec. 2011.

[32] NVIDIA, *CUDA SDK Code Samples v4.0*, Mar. 2016.

[33] Y. Oh, K. Kim, M. Yoon, J. Park, Y. Park, W. Ro, and M. Annavaram, "Apres: improving cache efficiency by exploiting load characteristics on GPUs," *ACM/IEEE International Symposium on Computer Architecture*, June 2016.

[34] M. Rhu, M. Sullivan, J. Leng, and M. Erez, "A locality-aware memory hierarchy for energy-efficient GPU architectures," *IEEE/ACM International Symposium on Microarchitecture*, Dec. 2013.

[35] T. Rogers, M. OConnor, and T. M. Aamodt, "Cache-conscious wavefront scheduling," *IEEE/ACM International Symposium on Microarchitecture*, Dec. 2012.

[36] T. Rogers, M. O'Connor, and T. Aamodt, "Divergence-aware warp scheduling," *IEEE/ACM International Symposium on Microarchitecture*, Dec. 2013.

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

[38] N. Vijaykumar, G. Pekhimenko, A. Jog, A. Bhowmick, R. Ausavarungnirun, C. Das, M. Kandemir, T. Mowry, and O. Mutlu, "A case for core-assisted bottleneck acceleration in GPUs: enabling flexible data compression with assist warps," *ACM/IEEE International Symposium on Computer Architecture*, June 2015.

[39] 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," *IEEE/ACM International Symposium on Microarchitecture*, Oct. 2016.
[40] J. Wang and Y. Xie, "A write-aware STTRAM-based register file architecture for GPGPU," Journal on Emerging Technologies in Computing Systems, vol. 12, no. 1, July 2015.
[41] C. Wittenbrink, E. Kilgariff, and A. Prabhu, "Fermi GF100 GPU architecture," *IEEE Micro*, vol. 31, no. 2, Mar. 2011.
[42] P. Xiang, Y. Yang, and H. Zhou, "Warp-level divergence in GPUs: Characterization, impact, and mitigation," *IEEE International Symposium on High Performance Computer Architecture*, Feb. 2014.
[43] X. Xie, Y. Liang, X. Li, Y. Wu, G. Sun, T. Wang, and D. Fan, "Enabling coordinated register allocation and thread-level parallelism optimization for GPUs," *IEEE/ACM International Symposium on Microarchitecture*, Dec. 2015.
[44] M. Yoon, Y. Oh, S. Lee, S. Kim, D. Kim, and W. Ro, "DRAW: investigating benefits of adaptive fetch group size on GPU," *IEEE International Symposium on Performance Analysis of Systems and Software*, Mar. 2015.
[45] M. Yoon, K. Kim, S. Lee, W. Ro, and M. Annavaram, "Virtual Thread: maximizing thread-level parallelism beyond GPU scheduling limit," *ACM/IEEE International Symposium on Computer Architecture*, June 2016.
[46] H. Zhang, X. Chen, N. Xiao, and F. Liu, "Architecting energy-efficient STT-RAM based register file on GPGPUs via delta compression," *ACM/EDAC/IEEE Design Automation Conference*, June 2016.