A Survey of Techniques for Architecting and Managing GPU Register File

Sparsh Mittal

Abstract—To support their massively-multithreaded architecture, GPUs use very large register file (RF) which has a capacity higher than even L1 and L2 caches. In total contrast, traditional CPUs use tiny RF and much larger caches to optimize latency. Due to these differences, along with the crucial impact of RF in determining GPU performance, novel and intelligent techniques are required for managing GPU RF. In this paper, we survey the techniques for designing and managing GPU RF. We discuss techniques related to performance, energy and reliability aspects of RF. To emphasize the similarities and differences between the techniques, we classify them along several parameters. The aim of this paper is to synthesize the state-of-art developments in RF management and also stimulate further research in this area.

Index Terms—Review, classification, GPGPU, GPU, register file, reliability, performance, power management, non-volatile memory, embedded DRAM (eDRAM).

1 Introduction

Driven by the ever-increasing performance demands, the architecture of GPU has evolved significantly in the last decade. Modern GPUs execute tens of thousands of threads for achieving high throughput and hiding memory latency. To enable efficient context-switching between these threads, large size register file (RF) is required. Scarcity of RF resources can harm performance by limiting the occupancy of GPU kernels [1]. Also, when the register requirement of a thread block exceeds the available capacity, some variables need to be allocated in local memory (termed as 'register spilling' [2]) which leads to performance penalty.

Driven by these requirements, the size of RF has increased significantly in recent GPU generations. This fact is confirmed by Table 1 which shows the size of RF and L1/L2 caches in recent GPUs (CC = compute capability). From the table, it is also clear that on GPUs, size of RF is much larger than that of L1 and L2 caches. Due to its large size and design with high-performance leaky transistors [3], RF contributes significantly to the GPU power consumption, for example, 17.2% and 13.4% of total dynamic power in Quadro FX5600 (CC = 1.0, 32KB RF) and GTX 480 (CC = 2.0, 128KB RF) GPUs is attributed to RF, respectively[1] [4]. Clearly, RF management plays a crucial role in meeting performance targets and area/power budget constraints in GPUs.

TABLE 1
Size of L1 cache[2], L2 cache and RF on NVIDIA GPUs [2, 5–9] (All sizes are in KB)

	Architecture	CC	L1 size per SM	L2 size	RF size per SM	# of SMs	Total RF size
G80	Tesla	1.0	None	None	32	16	512
GT200	Tesla	1.3	None	None	64	30	1920
GF100	Fermi	2.0	48	768	128	16	2048
GK110	Kepler	3.5	48	1536	256	15	3840
GK210	Kepler	3.7	48	1536	512	15	7680
GM204	Maxwell	5.2	48	2048	256	16	4096

By comparison, CPUs primarily focus on optimizing latency in serial applications, and hence, they execute at most few tens of threads and allocate a large portion of chip 'real estate' to caches. For this reason, CPUs have large caches and much smaller RF, for example, Intel's 32nm Itanium 9560 processor has 32MB L3 cache and 22KB integer RF and 20KB floating point (FP) RF (1.375KB integer RF and 1.25KB FP RF per thread for 16 threads) [10]. Thus, due to the fundamental differences between CPU and GPU architecture and RF sizes, conventional CPU RF management techniques cannot be retrofitted for GPU RF. Given the unique opportunities and challenges in architecting GPU RF, novel techniques are required to address performance bottlenecks and leverage the full potential of RF in improving performance. Several techniques have been recently proposed to fulfill this need.

In this paper, we present a survey of techniques for designing and managing GPU register file. Fig-

• *Support for this work was provided by U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research.* *The author is with the Future Technologies Group at Oak Ridge National Laboratory, Oak Ridge, Tennessee, USA 37830.*
E-mail: mittals@ornl.gov

1. The reduced fraction of RF power in GTX 480 despite larger RF size is due to increased contribution from other GPU components.

2. In Fermi and Kepler, 48KB is the maximum size of L1 cache. Also, in GM204, 48KB is the size of unified L1/texture cache.

ure 1 shows the overall organization of this paper. We begin with a brief background on opportunities and obstacles in managing RF (Section 2). Sections 3 through 5 classify RF management techniques in three categories. Section 3 discusses techniques for designing RF using high-density and low-leakage memory technologies. Sections 4 and 5 review techniques for improving performance/energy efficiency and RF reliability, respectively. Section 6 concludes this paper with a discussion of future challenges.

Paper organization

- **§2 Factors and tradeoffs in RF management**
  - §2.1 Achieving high energy efficiency
  - §2.2 Achieving high bandwidth and throughput
  - §2.3 Achieving high capacity
  - §2.4 Achieving high resource-utilization
  - §2.5 Achieving a balance with other objectives
  - §2.6 Achieving high reliability
  - §2.7 GPU evolution trends
- **§3 Architecting RF using Novel Memory Technologies**
  - §3.1 A background on memory technologies
  - §3.2 An overview and key ideas of techniques
  - §3.3 Architecting eDRAM based RF
  - §3.4 Architecting STT-RAM based RF
  - §3.5 Architecting hybrid SRAM-STTRAM based RF
  - §3.6 Architecting DWM based RF
  - §3.7 Architecting S/D based RF
  - §3.8 Architecting hybrid CMOS-TFET based RF
- **§4 Performance and Energy Efficiency Improvement Techniques for RF**
  - §4.1 An overview and key ideas of techniques
  - §4.2 Techniques for reorganizing RF architecture
  - §4.3 Techniques for RF access scheduling
  - §4.4 Techniques for managing RF power consumption
  - §4.5 Techniques for register sharing
  - §4.6 Techniques for leveraging redundancy in GPU execution
  - §4.7 Approximation techniques for RF
  - §4.8 Techniques for using RF as cache or shared memory
- **§5 Reliability Improvement Techniques for RF**
  - §5.1 Mitigating soft-error impact on RF
  - §5.2 Mitigating NBTI impact on RF
  - §5.3 Mitigating process variation impact on RF
- **§6 Conclusion and Future Outlook**

Fig. 1. Overall organization of the paper

To keep a balance between brevity and breadth, in this paper we concentrate on innovations focused on RF and not just other GPU components or GPU as a whole. Since different works use different evaluation approaches and applications, we focus on their key ideas and present selective numerical results to show the quantitative improvements. We hope that this survey will be useful for computer architects, designers and researchers[3].

2 Factors and Tradeoffs in RF Management

Figure 2 shows the RF architecture in Fermi GPUs. In what follows, we present some details on RF architecture. We also highlight the crucial importance of managing RF and discuss several factors/challenges which need to be accounted for while designing RF management policies. For more details on GPU architecture and power modeling, we refer the reader to previous work [3, 4, 12–15].

3. We use the following acronyms frequently in this paper: domain wall memory (DWM), FP unit (FPU), most significant bit (MSB), multi-level cell (MLC), negative bias temperature instability (NBTI), non-volatile memory (NVM), process variation (PV), read/write (R/W), single instruction multiple data (SIMD), special function unit (SFU), spin transfer torque RAM (STT-RAM), SRAM-DRAM hybrid memory [11] (S/D memory), streaming multiprocessor (SM), tunnel field-effect transistor (TFET).

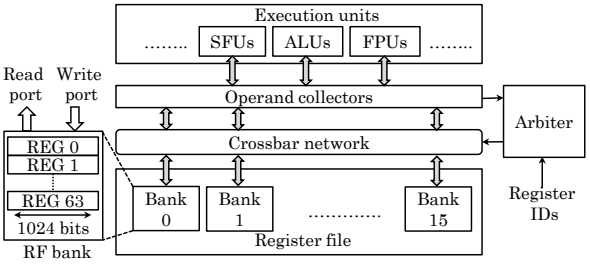

Fig. 2. RF architecture in Fermi GPUs [4]

2.1 Achieving high energy efficiency

For achieving small latency, RF must be designed with high-speed transistors [3], however, these transistors also dissipate large amount of leakage power. Also, GPU RFs are very wide: every 1024b RF entry provides 32b operand to all 32 threads in a warp. These, along with the large size of RF and physical distance of RF from execution unit make power consumption of RF a large fraction of overall power consumption. For example, Lucas et al. [14] observe that RFs contribute 8.7% of static power and 16.8% of dynamic power in a single GT 240 (CC = 1.2) core. Goswami et al. [15] note that RF leakage and dynamic power consumption is 17% and 44% (respectively) of the total core power in GTX 470 (CC = 2.0). Atoofian [16] observes the RF power consumption to be 16% of total power budget for a Fermi GPU.

Lim et al. [3] note that RF contributes 12% and 7% (respectively) of the dynamic and total power consumption of GTX 580 (CC =2.0) for a compute-intensive benchmark. Mao et al. [17] show that RF may be accessed 10 to 30 times more frequently than the L1 cache and thus, the dynamic power of RF may be up to 6 times its leakage power. Clearly, managing RF power consumption is crucial to stay within TDP (thermal design power) limit and optimize battery life in mobile GPUs [18].

2.2 Achieving high bandwidth and throughput

To provide high throughput, RF needs to allow multiple concurrent accesses, e.g., a fused-multiply-add operation which has 3 inputs and 1 output, can be performed in one cycle on Fermi GPU [4] and thus, RF is expected to allow (at least) four register accesses in every clock cycle [12]. Since multi-ported RF incurs high overhead, RF in existing GPUs uses multiple banks and operand collectors to provide high-bandwidth, for example, the 64KB RF in GT200 is likely to have 64 logical banks [12]. However, increasing the number of banks incurs high overhead while providing diminishing performance returns. For example, Jing et al. [19] show that for an RF size of 128KB per SM, increasing the number of banks from 16 to 32 does not significantly improve performance although it incurs high area/power/latency overheads. To provide the illusion of a multiported RF, GPUs also use operand collectors which access multiple single-ported register

banks and buffer the data. This, however, comes with a latency overhead.

2.3 Achieving high capacity

Given the requirement of large RF capacity and limited area budgets, novel approaches are required for architecting RF. For example, use of high density domain wall memory (DWM) can allow doubling the capacity of registers, cache and shared memory within same area/power budgets [20]. However, these memories also have their own limitations, for example, Tan et al. [21] show that due to the high write latency/energy of STTRAM, an STTRAM-only RF design leads to higher execution time and dynamic energy consumption compared to an SRAM RF. Clearly, novel architectural techniques are required for enabling their use in RF design (refer Section 3).

2.4 Achieving high resource-utilization

The RF resources are generally over-provisioned to meet the peak performance targets, however, their average utilization remains low [22–24]. For example, on use of even one register in a program context, the compiler allocates thousands of registers. Similarly, due to bank conflicts or long-latency memory stalls, several registers remain idle. Prudent management techniques can avoid such inefficiencies by virtue of runtime adaption.

2.5 Achieving a balance with other objectives

Due to limited area/power budgets, arbitrarily increasing RF size is infeasible since it reduces the budget for other components, e.g. cache, shared memory, etc. [25]. Also, the use of registers needs to be carefully balanced with other optimization considerations, for example, if a kernel uses too many registers, the total registers used by all warps can exceed the number of available registers [2]. To avoid this, the number of warps needs to be reduced which harms performance. Clearly, blindly increasing RF size or utilization is likely to be suboptimal or even infeasible.

2.6 Achieving high reliability

RF reliability can be threatened due to several reasons. Due to its large size, RF is susceptible to PV and becomes a bottleneck in deciding core frequency. Due to PV in RF, the GPU frequency can degrade by up to 40% compared to the GPU free of PV [26]. Due to the variation in the utilization of different registers, some registers may see much more NBTI-stress than others and they may age faster than others, thus, reducing the lifetime of the entire chip [22]. With ongoing process scaling, the charge required to flip a bit has been decreasing and hence, SRAM-based RF is becoming increasing susceptible to particle-strike induced soft-errors [27].

2.7 GPU evolution trends

Due to fast evolving GPU architecture, some of the techniques designed for one GPU model may not work for or be applicable to other models. For example, the technique of Falch et al. [28] utilizes shuffle instruction which has been introduced in Kepler GPU only [6]. Also, since the complete information about microarchitecture of commercial GPUs is not publicly disclosed, researchers typically infer these values by microbenchmarking [12] or from patents [4]; this, however, may not be fully accurate. These factors present a challenge in design of effective RF management policies.

3 Techniques for Architecting RF using Novel Memory Technologies

GPU RF has been conventionally designed using SRAM. However, SRAM has low-density and high leakage-power consumption. To address these limitations, several novel memory technologies have been explored for designing RF. We first provide a background on these technologies and then discuss several techniques for enabling their use in designing RF.

3.1 A background on memory technologies

Table 2 shows an overview of relative strengths and limitations of different memory technologies used for designing RF. We now briefly discuss their working and relevant properties and refer the reader to previous work [29, 30] for more details.

eDRAM: eDRAM is a capacitor-based DRAM which can be integrated on the same die as the processor. Two widely used eDRAM cell designs are 1T1C and 3T1D [32]. EDRAM has higher density and lower leakage than SRAM. Its retention period is in the range of tens of microseconds and due to this, its refresh overhead becomes high.

STT-RAM: STT-RAM stores data in a magnetic tunnel junction (MTJ). While its read latency/energy are comparable to that of SRAM, its high write latency/energy present a challenge in its use. Both STT-RAM and DWM are non-volatile and can store multiple bits in a cell, called multi-level cell (MLC).

DWM: DWM operates by controlling domain wall motion in ferromagnetic nanowires. The read/write latency of DWM is comparable to that of SRAM, and thus its write-performance is superior to that of STT-RAM. In both DWM and S/D memory, multiple register contexts (r-contexts) exist of which one is active which can be accessed with low latency. Shift operations are required to access other r-contexts and thus, these memories tradeoff data-accessibility for efficiency. The difference between DWM and S/D is that in DWM, the switching latency/energy vary based on the distance between the active and requested r-context, whereas for S/D memory they are fixed.

TABLE 2
A comparison of memory technologies [11, 17, 29, 31]

	Density	Leakage energy	Speed (R/W)	Access energy (R/W)	Access any bit	Refresh required	Main challenge
SRAM	Low	High	Very fast	Low	Yes	No	High leakage
eDRAM	High	Low	Fast	Medium	Yes	Yes	Refresh operations
STT-RAM	Very high	∼0	Fast/slow	Low/High	Yes	No	Write operations
DWM	Very high	∼0	Fast	Low/High	No	No	Context-switching
S/D	High	Low	Fast	Low	No	Yes	Context-switching

For DWM and S/D, access speed and energy also depend on number of bits or contexts per cell and the current active context.

S/D memory: Yu et al. [11] propose an SRAM-DRAM memory array where N DRAM branches (2N 1T1C DRAM cells) are integrated into each SRAM cell. This memory allows local copying of data between SRAM cell and one selected DRAM branch within a cell. The SRAM cell stores the active context which can be externally accessed. A DRAM branch cannot be externally accessed and hence, for accessing it, its value must be copied into SRAM. Thus, DRAM latency is hidden by only exposing SRAM to the core. Note that in S/D memory, only SRAM cell is externally accessible, whereas in SRAM-STTRAM hybrid memory [21], both SRAM and STTRAM registers are externally accessible.

CMOS and TFET: CMOS and TFET are different transistor types. MOSFETs switch by modulating thermionic emission over a barrier and CMOS is designed using symmetrical and complementary pairs of p- and n-channel MOSFETs. By comparison, TFETs switch by modulating quantum tunneling through a barrier. The sub-threshold slope of TFET is steeper than that of CMOS and hence, at low voltages TFET provides higher performance and ultra low leakage compared to CMOS. At high voltages, however, CMOS provides superior performance and this tradeoff presents an opportunity to bring their best together.

3.2 An overview and key ideas of techniques

Table 3 classifies the works based on memory technologies used by them for RF design. We now summarize some key ideas used by different works that use alternative memories for designing RF (also see Section 4.1). Their discussion follows.

1. Diverse register/cell properties: All three: process variation [26, 32], use of hybrid memory designs [21, 33] and use of MLC STT-RAM with hard/soft bits [34] lead to registers/cells with different latency/energy properties, and hence, management techniques for them show similarities. However, they are fundamentally different. PV occurs unintentionally due to manufacturing limitations and can cause random variations whereas hybrid memory or MLC are used intentionally and lead to well-understood differences.

2. Incremental register allocation: In most techniques, all registers are available for the applications to use. By comparison, in some techniques, only selected registers are initially employed to prioritize their use, e.g. registers designed with soft-bits in MLC STT-RAM [34], registers in fast-banks in PV-affected RF [26] and private registers in a shared RF architecture [35]. Remaining registers (e.g., registers designed with hard bits, registers in slow-banks and shared registers, respectively) are used only if the demand exceeds the available capacity.

3. Dynamically deciding register storage location: Some techniques use compiler to perform lifetime analysis of registers [1, 21, 36–38]. Based on a register's lifetime, it may be mapped to SRAM or STT-RAM [21] and to main RF or additional structures [39], etc. The access frequency of a register can decide whether it is stored in soft/hard bit [34] and instantaneous memory contention may decide whether a register is allocated in CMOS or TFET-based storage [33]. To reduce memory contention, some threads can be slowed down by mapping their registers to TFET cells [33] or PV-affected slow banks [26].

4. Refresh management in eDRAM-based RF: Fine-grain refresh schemes can avoid refreshing better-than-worst cells at a (higher) rate determined by the worst cell [32]. Based on criticality of a bit (e.g., MSB or non-MSB), it can be refreshed at regular rate or reduced rate [40]. Also, to avoid delaying regular accesses, refreshes can be scheduled when a bank is idle [32, 38]. Based on register lifetime, refresh operations to dead registers can be avoided [38].

5. Reducing access latency in NVM-based RF: STTRAM-based RF can use concurrent R/W operations for improving throughput [41, 42] and DWM-based RF can use speculative shifting for hiding shifting latency [16, 20].

6. Reducing accesses to RF: Write-back buffers can be used to coalesce the accesses [11, 15, 17, 20, 34, 41, 42] and this is especially useful for reducing writes to NVMs. Differential write scheme has also been used for reducing write operations [15].

7. Using prediction approach: Some techniques work by predicting *future* values of memory stall latency [33] and register access [16] based on their *past* values.

3.3 Architecting eDRAM based RF

Jing et al. [32] design eDRAM-based RF and present mechanisms to manage its refresh overhead. A naive

TABLE 3
Memory technology or cell type used for RF design

Classification	References
eDRAM memory	[32, 38, 40]
STT-RAM memory	[15, 34, 41, 42]
Hybrid SRAM-STTRAM memory	[21]
Domain wall memory	[16, 17, 20]
SRAM-DRAM (S/D) memory	[11, 20]
Hybrid CMOS-TFET design	[33]
Radiation-hardened cells	[43]

refresh mechanism blocks all the banks for refresh and at this time, RF pauses regular operation which harms performance. They propose a fine-grain refresh mechanism which works by refreshing idle banks. A bank remains idle if an instruction accesses only few banks. Also, a bank conflict serializes RF accesses and during this stall period, remaining banks remain idle. Their mechanism preferentially chooses those idle banks for refresh which are nearing the end of their retention period. This mechanism hides refresh overhead with periods of inactivity. In absence of idle periods, a mandatory refresh is issued to avoid data loss. Since this mechanism requires one refresh counter for each register entry, they present a second mechanism to reduce this overhead. This mechanism uses a single global counter. When this counter nears expiration, the banks are refreshed one after another and every time one entry in a bank is refreshed. Compared to a naive mechanism, this mechanism blocks only one bank at a time and remaining banks can function normally. Also, the bank being refreshed becomes free much sooner than in the naive mechanism.

Since read accesses are destructive in 1T1C eDRAM cells, they also present a technique for addressing this. They double the number of banks and also divide the warps in even/odd groups based on their ID. Even/odd warps are mapped to even/odd banks, respectively. The warp scheduler issues odd and even warps at successive cycles. When an odd warp instruction accesses odd bank group, read access and write-back (for restoring the capacitor to the original value) happen in one cycle each. An even warp instruction is issued to even bank group after odd warp issue slot to avoid bank conflict. Thus, the penalty of destructive reads is alleviated by interleaving even/odd warps. Their mechanisms improve energy efficiency compared to SRAM RF.

Jing et al. [38] present a refresh mechanism for eDRAM-based RF. They note that data values in eDRAM cells remain unaffected as long as the refresh is scheduled within the retention period. Based on this, if a refresh operation conflicts with the regular RF access, their technique postpones the refresh and makes a second attempt for refresh within retention period and without blocking regular access. Since on a bank conflict, remaining banks stay idle, a second refresh attempt is made during this time period. If a bank is not found to be idle for the entire retention period, a refresh operation is forced to avoid data loss. For this, their technique counts the number of RF entries for which refresh is postponed and if this exceeds a threshold, a refresh operation is enforced. Thus, this technique aims to utilize bank idleness due to unbalanced accesses to different banks for hiding refresh penalty. They further note that due to over-provisioning of RF resources, many registers may not be active during warp execution and several others may be unused and hence hold garbage data. Based on this, they use a 'liveness bit' with each register. Based on the register liveness information from the compiler, this bit is set on a write access to the register with subsequent read access and is reset on the last read from the register. A register value needs to be refreshed only if its liveness bit is 1 indicating future reuse. Using this, refresh operation to dead entries can be avoided. They show that compared to an SRAM-based RF, use of their technique with eDRAM-based RF achieves large improvement in energy efficiency.

3.4 Architecting STT-RAM based RF

Goswami et al. [15] use STT-RAM for designing RF and propose mechanisms to reduce writes to it. On any register write, their first mechanism updates only the changed register arrays and skips writing to unchanged arrays, instead of writing the whole register. Since GPGPU applications generally modify neighboring memory cells of a register, they organize a K-bit register word in P arrays of Q-bits/array ($K = P \times Q$) and provision write to the entire Q-bit array on a change in any bit of the array. Since STT-RAM read latency is smaller than the write latency, an additional read operation to find changed arrays does not harm performance. Their second mechanism uses an SRAM write-buffer for coalescing writes issued by multiple threads to a bank. They show that their mechanisms reduce the energy consumption of RF.

Liu et al. [34] use MLC STT-RAM for designing RF and propose architectural techniques to manage this. Every RF entry with 2048 bits is designed using 1024 MLC STT-RAM cells, which can be seen in terms of 1024 hard-bits (slow row) and 1024 soft-bits (fast row). To avoid overflow of writebacks, a write-buffer is used in each bank. Since the access latency and energy of hard bit is much larger than that of soft bit, they propose an address remapping scheme. Using compiler, the access frequencies of all registers used for a kernel execution are recorded. Based on this, the decision to place a register in soft/hard-bit row is taken. The hard-bit rows are used only when more than half of the total register capacity is required and in such case, frequently-accessed and remaining registers are stored in soft-bit and hard-bit rows, respectively. To further reduce the impact of long STT-RAM write latency, they propose a warp scheduling scheme which preferentially issues ready warps that

access unoccupied register banks. They demonstrate that their technique enhances GPU performance and energy efficiency.

Li et al. [41] note that the inactivity period of registers is much longer than their activity period (e.g., 11K cycles vs. 230 cycles). This provides opportunity to reduce their leakage power consumption by designing RF with NVM instead of SRAM. They propose an STT-RAM RF design with two SRAM write buffers since a single write buffer (e.g., as used in [15]) may not capture spatial locality of RF access. One buffer stores data for currently active warp whereas other buffer writebacks data to STT-RAM RF when its warp becomes pending due to long latency accesses. These buffers coalesce write accesses to RF and serve read accesses if the requested register is stored in the buffer. The STT-RAM RF has two banks and registers of warps with even/odd warp IDs are mapped to 0/1 banks, respectively. This allows simultaneous read/write access and alleviates the impact of long STT-RAM writeback latency of one warp on another active warp. To reduce the leakage power of SRAM buffers, they use power gating. For this, they note that out of 64 buffer entries, only around 16 entries are typically used. Hence, they keep the first 16 entries always ON and keep the remaining 48 entries power gated. When more than 16 entries are required, remaining 48 entries are all turned ON. When all the 64 entries are written-back to STT-RAM RF, the 48 entries are again power-gated. They show that their approach reduces RF power consumption.

3.5 Architecting hybrid SRAM-STTRAM based RF

Tan et al. [21] propose an SRAM-STTRAM hybrid RF design and architectural techniques to bring together the benefits of SRAM and STT-RAM. By virtue of using magnetic storage, STT-RAM is immune to particle strike-induced soft errors [29], however, write to it incurs high overheads and opposite is true for SRAM. Their first technique uses compiler to find the lifetime (i.e., instruction-count between the write and last read) of different register values. They observe that most values have short-lifetime and they do not significantly affect RF soft-error vulnerability, whereas the remaining (~20%) long-lived values contribute significantly (~90%) to the vulnerability. Based on this, at compilation time, long-lived (e.g., for more than 10 instructions) register values are mapped to STT-RAM to protect them from soft-errors, and short-lived values are mapped to SRAM. This also reduces writes to STT-RAM and addresses its high write latency/energy issue. To further reduce STT-RAM write penalty, their second technique works on the observation that most RF writes are narrow-width, such that out of 32-bits, the higher order 16-bits are zeros. Based on it, two narrow writes can share the bus bandwidth to reduce write overhead. Such write merging is performed only for writes from two warps that execute in two successive cycles and target different banks. They show that for negligible performance penalty, their techniques bring large reduction in energy consumption and RF soft-error vulnerability.

3.6 Architecting DWM based RF

Mao et al. [17] note that increasing the number of access ports in DWM reduces the shift distance at the expense of increased area, power and peripheral circuit latency overheads. They propose three schemes to alleviate the effect of shift latency in DWM-based RFs. To reduce the shift distance in accessing DWM, their *register remapping* scheme places the registers around the access ports of those banks, whereas the naive (default) mapping interleaves the registers of each warp across the RF banks consecutively. This remapping approach, however, is only effective for applications which have a large number of unused registers and cannot reduce shift latency of register-intensive applications. Their *warp scheduling scheme* preferentially issues a warp whose instruction generates RF requests with lowest distance to the current location of access ports. To avoid the interference effect of write requests in the scheduling scheme, they use a *write buffer*. Write requests are stored in the buffer and are issued to RF only when the corresponding register aligns with the access ports. Buffer overflow forces RF write irrespective of the alignment, however, such situation happens infrequently. Using this approach, their scheduling scheme needs to account for only read requests while calculating distance to access ports. They show that their technique improves performance and energy efficiency.

Atoofian [16] uses DWM to design RF and propose intelligent pre-shifting mechanisms to hide the shifting latency. His technique stores the registers in a bit-interleaved manner, such that instead of storing a 4B register in a 32-bit track, they store K 4B registers in 32 K-bit tracks. This reduces the access latency by allowing parallel access to all bits. He notes that threads in a kernel run same set of instructions and hence, the register ID of destination and source operands accessed by instructions in a GPGPU application is same, except when branch divergence happens. Thus, future register accesses can be predicted based on the past behavior. He proposes three policies which exploit data locality between threads in a warp (intra-warp), warps in an SM (intra-SM), warps in different SMs (inter-SM), respectively and use this to pre-shift the track head. To ensure timely pre-shifting, his technique chooses the length of RF tracks in a manner to fully hide the shift latency. Experimental evaluation shows that intra-SM policy provides better performance than inter-SM policy which, in turn, outperforms intra-warp policy. Also, intra-SM policy provides significant energy saving compared to SRAM RF.

3.7 Architecting S/D memory based RF

Since S/D memory does not support reading from/writing to different contexts, Yu et al. [11] present a technique to address this limitation. They use a write-buffer with each context which delays writebacks till this context is activated again. To refresh DRAM context, it is brought to SRAM and then copied back. Since re-reference interval of a context is expected to be smaller than the DRAM retention period, this refresh overhead remains negligible. The context-switching latency is hidden by RF-banking and background-loading of the contexts. They show that compared to an SRAM-only RF, their design provides significant area and energy advantages with only small performance loss.

Moeng et al. [20] present an RF architecture for integrating memory technologies with non-uniform access behavior, viz. S/D memory [11] and DWM. Every memory element stores K registers corresponding to K r-contexts. Concurrent access to these K registers cannot take place and thus, they form a multicontext group (MCG). When an instruction is committed, a write to RF may not complete if it differs from the active r-context. To address this, they use a write buffer and allow writes to preempt other commands. When the buffer becomes full, the system can proceed only when at least one write has finished. This scheme avoids write-related hazards. As for switching granularity, they consider multiple schemes: first, whole RF switches synchronously; second, every register in a bank switches together but different banks switch independently; and third, switching happens in only the MCG corresponding to a desired register. The third scheme reduces switching overhead and also leverages temporal locality of register accesses since an MCG moves out of existing context only if any of the other registers in the bank is accessed before next access to it. The third scheme also avoids preempting reads to other MCGs on a write access. They show that compared to the second scheme, the third scheme provides better performance.

Moeng et al. [20] further note that while waiting for RF reads to complete, instructions remain in operand collector. In their technique, when an instruction is placed into operand collector, preswitch command is issued to the MCG with register. In case of no ongoing access, the MCG starts switching to the requested r-context which helps in hiding the r-context switching latency.

3.8 Architecting hybrid CMOS-TFET based RF

Li et al. [33] present a hybrid CMOS-TFET based RF design. They note that memory contention due to a massive number of memory requests causes pipeline stall. To address this performance bottleneck, progress of some threads can be slowed down by using TFET-based registers. This delays their memory requests and also saves energy since at low supply voltages, the leakage power consumption of TFETs becomes very small. Since the memory stall latency can only be known after a request has been serviced, they predict this latency from that of the last memory request. Based on this, TFET-based registers are allocated to that thread to just hide the entire latency without losing performance. This is done using analytical modeling of thread stall time and TFET delay and it ensures optimal TFET register utilization for balancing performance and power. CMOS-based registers are allocated to threads requiring normal progress. They choose the ratio of TFET registers to total registers as the average ratio of warp stall time to total execution time. Their technique saves energy with small performance loss and provides a better balance of power and performance than power-gating scheme, drowsy scheme and using all TFET registers.

TABLE 4
Optimization objectives and management approaches

Classification	References
Energy	[1, 11, 16–22, 32–34, 36–41, 44–49]
Performance	[1, 17, 19, 20, 24, 26, 28, 34, 35, 46, 48, 49]
Approximate computing	[40]
Adapting warp scheduling scheme	[17, 18, 26, 34, 35, 37, 39]
Power/clock gating	[4, 18, 22, 36, 41, 44, 45, 47]
Use of drowsy state	[18, 45]

4 Performance and Energy Efficiency Improvement Techniques for RF

4.1 An overview and key ideas of techniques

Table 4 provides an overview and classification of different techniques. We now summarize some dominant ideas used by these techniques (also see Section 3.2).

1. Leakage energy management: Based on their data retention characteristic, the leakage energy saving approaches can be classified into state-preserving and state-destroying [50]. State-preserving (also known as 'drowsy') approach preserves the state of the block in low-leakage mode, whereas state-destroying approach (e.g. power gating) loses the data state in low-leakage mode. The former approach generally brings smaller reduction in energy than the latter. Several techniques have used these energy saving approaches (refer Table 4).

2. Improving RF utilization: To avoid under-utilization, registers can be shared across warps [1, 35, 36, 39]. Also, unused registers can be utilized for storing prefetched data [24] or as a shared memory pool accessible to all threads [28]. Further, storage for RF, cache and shared memory can be unified to allocate just suitable capacity to them [48].

3. Compiler-based techniques: Some techniques use compiler to track register usage pattern for find-

ing register lifetime [1, 18, 21, 36–38], register access frequencies [34, 39] and application's register requirement to avoid spills [48]. Also, compiler is used for reordering register declarations [35] and inserting prefetch instructions [24].

4. Action taken on long/short latency operations: Threads waiting for long-latency operations can be excluded from consideration by the scheduler [37] or their registers can be placed in drowsy mode [18]. SIMD lanes of inactive threads can be clock-gated during branch divergence [4].

5. Reducing RF accesses and contention: The accesses to RF can be reduced by caching/mapping some RF accesses in other structures [37, 39]. Some techniques use odd/even warp to bank mapping schemes [20, 32, 41] and a few others seek to mitigate bank-conflicts [19, 41, 42].

6. Exploiting redundancy: In presence of redundancy within and across warps, result reuse can allow reducing the computations [46, 49]. Also, performance and/or reliability can be improved by exploiting narrow values [21, 49] and using data compression [44].

4.2 Techniques for reorganizing RF architecture

Gebhart et al. [37] present two techniques to reduce accesses to and energy consumption of RF. Since register accesses show significant locality, they use a small register cache (RegCache) to capture hot register working set of active threads. This RegCache helps in avoiding writing of short-lived values to main RF. They observe that even a 6-entry/thread RegCache can reduce the accesses to main RF by nearly half. Further, by utilizing lifetime information from compile-time analysis, writeback from RegCache to main RF for those registers can be avoided which have seen their last read operation. They also propose a two-level scheduler which logically divides the threads into active and pending threads. The active threads are those issuing instructions or stalling on short latency (e.g., ALU or local memory) operations, whereas the pending threads are those stalling on long latency (main memory) operations. In any cycle, their scheduler only considers the active threads and this reduces its energy consumption. By combining the two techniques, further improvement can be obtained since the scheduler limits RegCache resource allocation to currently active threads and when an active thread stalls on main memory, its entry in RegCache can be flushed and both these greatly reduce the storage requirement of RegCache. Their techniques reduce RF energy consumption significantly.

Gebhart et al. [39] note that a limitation of RegCache is that values evicted from it consume RegCache read energy before being written to main RF. Also, tags and lookups are required in RegCache for it to track register names from the large-sized main RF. To address these, they propose a compiler-managed operand RF (ORF) design. Based on compiler information about register usage behavior, repeatedly or soon-to-be accessed values are mapped to ORF and values with limited temporal locality are mapped to main RF. Values which are persistent and show locality are stored in both ORF and main RF in the same instruction which avoids writeback on eviction. Also, ORF does not require tags or lookup operations since the operand storage location is ascertained at decode time. They further note that a large fraction of register values are read only once and a majority of them are read within three instructions after their generation. Storing these short-lived values in a small structure reduces system energy. To capture such values, they use a one entry/thread 'last result file' (LRF). This leads to a three-level register hierarchy, where LRF, ORF and main RF all have similar access latency but different access energy. They also present compiler algorithm to manage data allocation in these levels and efficiently sharing this hierarchy across the warps. This algorithm seeks to allocate maximum possible number of values in LRF first, then the same for ORF and finally it allocates values in main RF. They show that their compiler-based RF hierarchy management approach achieves larger RF energy saving than the hardware-only management approach [37].

Gebhart et al. [48] note that different GPU applications present different requirements of cache, shared memory and RF and their performance may be constrained primarily due to any of these three resources. Hence, using a fixed partition size for them leads to suboptimal use of on-chip storage. They present a memory architecture which unifies these three storage structures and permits flexible per-kernel size partitioning. To reduce the RF accesses and avoid bandwidth bottleneck in unified storage, they use software-managed RF and two-level warp scheduling [37, 39]. Also, a write-through cache is used to avoid holding dirty data in cache. The partitioning can be changed before the beginning of a kernel. Since RF and shared memory do not persist beyond thread-block boundaries and cache has no dirty data, the overhead of maintaining state on repartitioning is minimal. If different kernels of an application show similar memory footprint, a single memory partitioning determined at application start-time can be used without requiring repartitioning. As for determining the partition sizes, the size of shared memory is specified by the programmer, that of RF is determined by compiler to minimize register spills and the remaining storage is allocated to cache. By virtue of better utilization of on-chip storage, their technique reduces main memory accesses and improves performance and energy efficiency.

4.3 Techniques for RF access scheduling

Jing et al. [19] note that in GPGPUs, RF banks remain idle in some cycles and show conflicts in other

cycles, and these conflicts degrade the throughput by serializing accesses. They present a bank-stealing technique to balance RF bandwidth utilization by filling the idle banks with expected upcoming reads. Since in GPGPUs, generally multiple ready warps remain available at issue stage, their technique steals an idle bank at the present cycle for an operand to be fetched in next cycle. This alleviates an upcoming conflict which would have caused a pipeline stall. For this, their technique identifies warps to be issued in next cycle and looks for bank conflicts between operands of present and next warp instructions. They show that their technique improves performance and energy efficiency.

Wang et al. [42] use STT-RAM for designing RF and propose two mechanisms for addressing high write latency/energy limitations of STT-RAM. They note that STT-RAM uses current sensing scheme where sense amplifiers and write drivers are fully separated, in contrast with SRAM where the circuits of voltage sense amplifiers and write drivers are partially shared. Thus, in STT-RAM, two subarrays forming the memory array share sense amplifiers, which allows concurrent write and read accesses to two different subarrays in a single bank. Based on this, their first mechanism alters the arbitrator to allow non-conflicting read/write to the same RF bank for increasing parallelism. This mechanism improves performance compared to naive STT-RAM RF. Their second mechanism uses an SRAM write buffer to consolidate the writes issued by arbitrator, such that writes to RF happen only when the buffer becomes full. Write buffer entries for pending warps (i.e. waiting for long-latency operations) are preferentially replaced since only active threads access the RF, whereas pending warps will wait for a long time before becoming active again. This mechanism reduces writes to STT-RAM and helps in further improving energy efficiency of STT-RAM RF compared to SRAM RF.

4.4 Techniques for managing RF power consumption

Hsiao et al. [18] note that shading programs executed on mobile GPUs do not fully utilize the available registers. They propose a power-mode transition technique to save RF energy. They assume that every register bank is organized into partitions (e.g. 4 registers/partition). Since the first partition is frequently used for ensuring quick response, only drowsy scheme is used for this. In the beginning, the first partition is in drowsy mode and remaining partitions are power gated and hence, they use compiler information to insert wake up instructions for registers before their first use. Further, during long-latency memory operations, the register banks are switched to drowsy mode. Thus, unused partitions remain power-gated and short-term opportunity for power saving is exploited using drowsy mode. Since a frame processing rate of greater than 30 frames/second (i.e., real-time rendering) wastes battery life without improving user experience, they also propose a thread scheduling technique which works by estimating the number of threads required for a desired frame rate. To obtain admissible human perception, it brings the frame rate to 25 or 30 frames/second and power gates extraneous register banks. They show that use of both techniques leads to significant saving in RF leakage power consumption.

Abdel et al. [45] present techniques to save leakage and dynamic energy in RFs. In GPUs, several registers are not allocated by the compiler for program execution, and their leakage energy saving technique applies power gating to these registers at the beginning itself for the entire execution. Also, since the average time between successive accesses to remaining (i.e. allocated) registers ranges in hundreds (e.g. 800) of cycles, after any access, these allocated registers are at once placed into state-preserving drowsy state. Since branch divergence and limited parallelism lead to several idle threads in a warp, they identify these threads using the built-in active mask before an instruction is scheduled. Using this information, their dynamic energy saving scheme does not charge bit lines and word lines of registers of any idle thread in a warp. They show that their techniques reduce the RF energy consumption significantly.

Lee et al. [44] note that arithmetic difference between two successive thread registers in warp is generally small. This is because, all threads of a warp execute same instruction and hence, RF access occurs at warp granularity. Hence, the computations which depend on thread-index operate on register data that exhibit strong value similarity. Based on this, they propose a warp-level register compression scheme for saving GPU energy. For data compression, base-delta-immediate (BDI) algorithm [51] is used. For a banked RF, one register or one register bank is chosen as base for BDI algorithm and deltas from remaining registers or banks are calculated. This approach saves dynamic energy since due to compression, fewer register banks need to be activated on every warp-level register access. Also, leakage energy is reduced by saving register content in fewer banks and power gating the unused banks.

Leng et al. [4] note that during branch divergence inactive lanes consume power without performing useful work. To address this, they present a technique which clock-gates all SIMD lanes except those with active threads. This technique efficiently exploits small-latency branch divergence events. Interconnection network, operand collectors and execution units are gated at SIMD lane level and RFs are gated at bank level. They show that this technique is effective in reducing dynamic power.

Lim et al. [3] present a power-model for GPUs

which obtains initial data from McPAT for a GPU configuration and then adapts the models by comparison with empirical data. They note that many microarchitectural details such as cell-type of GPU RF is unknown and this presents difficulties in accurate estimation of power. Hence, they use the observation that GPU RF is heavily accessed and is optimized for fast access even at the cost of some leakage power dissipation. Based on these and calibration against the data from [37], they predict that RF is designed using 32-bank dual-ported (1 read/ 1 write) RAM blocks with high-performance cell type. Based on their power model, they find leakage and dynamic power consumption of GPUs and observe that RF contributes significantly to dynamic power consumption, especially for compute-intensive applications. They also study the effect of changing the maximum number of active blocks per SM which indirectly changes the effective register size. They find that for blackscholes benchmark, increasing the number of register size from 4096 to 24576 increases the IPC (instruction per cycle) and power consumption, however on going from 24576 to 32768 registers, the curves become relatively flat showing diminished returns.

4.5 Techniques for register sharing

Jeon et al. [36] note that GPU compiler reserves registers for every warp, however, these warps are scheduled at different time periods. Thus, after completion of execution of a warp, these allocated registers still consume power even though they are no longer used by any instruction. They propose sharing the registers across warps to avoid their under-utilization. They use compiler to obtain register lifetime values. Based on this, a register is freed from its physical space after last reference to it and this free register location is assigned to register of a different warp. For performing such architectural-to-physical register mapping, a hardware-based low-overhead register renaming approach is used. They show that their technique reduces the RF size requirement of the application and also saves energy by virtue of reducing live register space.

Yu et al. [1] note that while scarcity of RF resource limits occupancy of thread-blocks, only a fraction of registers allocated to a block are used at runtime. They present a register-sharing technique for improving performance and energy efficiency. Their technique disregards RF limitation while scheduling warps to each SM. Registers are dynamically allocated to and reclaimed from the warps at runtime. When the register demand of running warps exceeds the available capacity, some warps are temporarily suspended and their registers are stored in memory. The registers thus released are used by other running threads to make progress. When RF utilization again falls below its capacity, either suspended warps can resume execution (after loading their register data from memory) or new warps can begin execution. Their technique allows scheduling more warps within same RF capacity and allows scheduling kernels which require more RF resources than the available capacity.

Jatala et al. [35] present a technique which enables sharing of RF to allow launching additional thread blocks per SM for improving performance. For example, with 45K registers and each block requiring 10K registers, only 4 blocks can execute in a naive scheme. Their technique allocates 10K registers to three blocks and shares 15K registers between two blocks, such that a pair of warp from those blocks gets 0.5K registers exclusively and 0.5K registers are shared between them. If a warp from one block accesses a shared register, it gets exclusive access to the shared 0.5K registers and the warp from another block can access the shared registers only after the original warp has finished. Their technique also limits the number of additional blocks launched since over-sharing can cause severe contention and inhibit any progress in those threads.

They further propose a warp scheduling technique for effective resource utilization. For two blocks that share registers, if any warp of first block waits on second block for shared registers, the second block and its warps are termed as owner block and owner warps, respectively and the first block/its warps are termed as non-owner block/warps, respectively. This technique schedules warps in following order: shared owner, exclusive (i.e, unshared), and shared nonowner. Thus, owner warps can complete sooner allowing dependent non-owner warps to progress and these non-owner warps hide stalls when no other warps are executing. Further, to allow non-owner warps to complete many instructions before accessing shared registers, their technique unrolls and reorders the register declarations. Also, since additional blocks can cause memory congestion, their technique records the number of memory instructions from non-owner warps and when the stall becomes excessive, their technique reduces the probability of memory instructions in these warps. They show that their technique improves performance significantly.

4.6 Techniques for leveraging redundancy in GPU execution

Xiang et al. [46] note that due to several code features, e.g., loops, initialization from constant values or same address, etc., the input and output values of different threads in a warp *and* different warps may be same. They refer to such instructions as intra- and inter-warp uniform vector instructions (UVIs), respectively. They propose techniques to leverage such redundancy for improving performance, energy efficiency and reliability. Their first technique adds a flag to each vector register to identify and record intra-warp UVIs. If all

the source operands of an instruction have this flag set and no control divergence exists, this instruction is marked as intra-warp UVI. For such an instruction, only one thread in a warp performs computation which is reused and stored by all the remaining threads. Their second technique uses a small scalar RF. Accesses to uniform vectors are served from scalar RF instead of vector RF, which lowers the energy consumption.

Their third technique exploits inter-warp redundancy. It uses an instruction reuse buffer (IRB) which stores results at warp-level. If an instruction is found to be already present in IRB, its execution is skipped and the result stored in IRB is directly written to destination register. Only instructions detected as inter-warp UVIs access IRB which simplifies its management and avoids the need of storing/comparing operands of all the threads of a warp. These techniques boost performance and save energy. For reliability enhancement, for intra-warp UVIs, two threads are used to perform redundant computations and save their two results and for inter-warp UVIs, the IRB is protected with parity bits. This approach improves reliability coverage of RF and ALU against hardware errors without harming performance.

Gilani et al. [49] propose two techniques for improving the utilization of computation resources. They note that in most (e.g. up to 50%) of instructions, same computations happen in multiple threads, due to factors such as inherent redundancy in data, processing constants, same operations being performed across threads of a warp, etc. Their technique detects instructions for which the results of all threads of a warp are same. Such instructions are issued to a separate scalar pipeline which keeps its registers in a separate scalar RF. By virtue of avoiding duplicated calculations and freeing the vector pipeline for executing another instruction in parallel, this technique improves both performance and energy efficiency. They further note that for several instructions, the operands and results values are narrow, e.g. less than 16 bits for 32-bit datapath. Based on this, they divide the 32-bit datapath into two 16-bit portions which allows issuing and executing up to two narrow-instructions in every cycle for improving performance. Alternatively, by running only one narrow instruction, the RF access energy can be reduced.

4.7 Approximation techniques for RF

Jeong et al. [40] note that for several multimedia applications, a small number of errors do not degrade user experience [52]. Also, a tradeoff between output quality and power consumption can be achieved by scaling the precision of floating values. They use an eDRAM-based RF and individually control the refresh rate of every 32-element register entry. An entry with precise value and the higher-order bits (e.g. upper half word) of an approximate value are both refreshed at regular rate. The latter helps in achieving acceptable quality since the higher-order bits affect output quality more than the lower-order bits [52]. The lower-order bits are refreshed at less than regular rate, which saves refresh energy. They enhance GPU pipeline to automatically identify FP values by tracking register operands of each FP instruction. This avoids the need of programmer annotation for identifying approximable values. They show that their technique saves significant refresh energy for both single- and double-precision FP data.

4.8 Techniques for using RF as cache or shared memory

Lakshminarayana et al. [24] note that due to their irregular control flow and data-dependent memory access pattern, graph algorithms achieve low efficiency on GPUs. They propose a prefetching scheme for hiding memory latency in graph algorithms. Dependent loads where one load depends on other are common in graph algorithms and for such load pairs, their technique identifies the target loads and prefetches data into currently unused registers. The prefetch distance (i.e. how well in advance a prefetch instruction is inserted) can be fixed or adapted on a per-loop basis such that for K unused available registers, a prefetch distance of K can be used, and if sufficient registers are not available, the distance is reduced. Due to very large number of threads, GPU caches see much more severe contention than the CPU caches and hence, storing prefetched data into registers instead of cache avoids their premature eviction. They show that their technique improves performance for memory-intensive graph applications.

Falch et al. [28] present a technique which combines registers from multiple threads to form a pool which can be employed as user-managed last level cache, similar to shared memory but with smaller latency. Threads can access registers of other threads using shuffle instruction [6] at a latency higher than that of their own registers but lower than that of shared memory. The limitation of their technique is that only the threads in a single warp can form a shared pool. Also, sharing provides benefit only when the threads have shared working set and thus, data stored in the pool can be reused by several threads. Further, the pool is read by only few threads which creates thread divergence. Also, management of this pool requires significant user-effort and hence, their technique is useful primarily for workloads with regular access pattern such as stencil computations. They show that compared to using shared memory, their technique improves performance in some applications and reduces it in others and the improvement depends on several factors, e.g., occupancy (number of thread blocks on an SM) and global memory efficiency, etc.

5 Reliability Improvement Techniques for RF

Reliability improvement techniques have been traditionally considered optional for GPUs due to their aggressively performance-oriented design and use in error-tolerant multimedia applications [27]. However, due to increased error rates, and growing use of GPUs for general-purpose applications, reliability techniques have become highly important for GPUs. Table 5 provides an overview of several techniques for improving reliability of RF. Some of these techniques use redundant execution for improving reliability [43, 46] and store a bit in radiation-hardened or normal cell based on its criticality [43]. We now discuss these techniques.

TABLE 5
Vulnerabilities addressed by different works

Classification	References
Soft-error	[21, 43, 53, 54]
NBTI-induced voltage degradation	[22]
Process variation	[26, 32]

5.1 Mitigating soft-error impact on RF

Farazmand et al. [54] compute the architectural vulnerability factor (AVF) for GPU RF. Note that AVF shows the average time when a circuit-level soft-error gets propagated to other processor components. The AVF value for GPU RF is observed to be much smaller than that typically observed in CPU RF, which can be attributed to low utilization of GPU RF. They propose an 'AVF-util' metric which measures the AVF of only the allocated (utilized) registers and this metric was found comparable with AVF of CPU RF. Overall, their study suggests GPU-specific soft-error studies for effective design of protection mechanisms and selection of most vulnerable components.

Tan et al. [53] study soft-error vulnerability of several GPU structures, e.g. RF, SMs, warp scheduler. The vulnerability is found to vary with application attributes, e.g., branch divergence, memory access behavior. In different applications, different per-block resources (e.g. RF, shared memory, etc.) become bottleneck in deciding the number of blocks allocated to an SM at a time. The bottleneck resource itself is utilized maximally and hence, it shows high vulnerability, whereas the remaining underutilized resources show error-tolerance behavior. For the structures studied by them, IPC/AVF and AVF were found to be uncorrelated. As for the impact of microarchitectural policies, they find that dynamic warp formation scheme (which handles branch divergence by regrouping threads from different warps that branch to the same destination) and use of higher number of per-SM threads lead to increased soft-error robustness of RF, whereas warp scheduling policy has negligible effect on RF vulnerability.

Palframan et al. [43] observe that for several FP intensive GPU programs, large magnitude errors can get further amplified to significantly degrade the output, whereas small magnitude errors have inconsequential impact on the output. They present a precision-aware technique to reduce large magnitude errors in RF and execution logic. RF is protected from large errors by storing the register MSBs in hardened cells since applying logic hardening to all gates is costly. For protecting significand value in FMA (fused multiply add) FP unit, they use selective hardening approach. The gates for hardening are chosen based on the relative magnitude of error that a soft-error in those gates can produce. Further, only the MSBs of the significand computation are verified through redundant execution on a light-weight checker circuit. They show that for the same overhead, compared to a magnitude-unaware selective protection approach, their approach reduces the mean error magnitude.

5.2 Mitigating NBTI impact on RF

Namaki-Shoushtari et al. [22] note that for several applications, the utilization of RFs is non-uniform. They present a technique which improves the lifetime of RF by uniformly spreading the NBTI stress and power-gating the most-stressed banks. In GPUs, RF allocation happens at the granularity of workgroup (a workgroup contains up to four wavefronts) and thus, within a workgroup, the RF regions show same aging behavior. During RF allocation, their technique proactively chooses RF regions with less aging to distribute the aging uniformly over entire RF. At the completion of a workgroup, the RFs are power gated which places them in recovery mode since power-gating reduces NBTI stress by providing sleep states to the circuit. They show that their technique reduces the threshold voltage degradation due to NBTI and also saves leakage power.

5.3 Mitigating process variation impact on RF

Tan et al. [26] note that due to their large size, RF is susceptible to PV and it becomes a bottleneck in scaling core frequency which is determined by the slowest register speed [55]. They note that with current GPU RF floorplan (e.g., each RF bank holding 64 128B wide entries), the systematic effect of PV in vertical direction is much larger than that in the horizontal direction in every RF bank. Their first technique vertically classifies RF banks into subbanks and the latency of the slowest register determines that of the entire subbank. A two subbank per bank partition provides sufficient improvement and a more fine-grained partition does not provide additional gains. The fastest $N\%$ subbanks are classified as 'fast' and the remaining $100 - N\%$ are termed as 'slow'. For N = 70, the slowest among 70% fast subbanks determine the frequency and the 30% slow subbanks complete

an access in 2 cycles. They show that their *subbank-level* fast/slow partitioning technique provides nearly same frequency enhancement as a *register-level* partitioning technique.

A limitation of this technique is that presence of even one slow subbank makes the bank as slow which determines the bank latency. To address this, their second technique logically combines similar type (i.e., fast/slow) subbanks to make a new bank. Since the latency of both subbanks becomes same, an access mapped to a fast bank can complete in 1 cycle. Their third technique works on the observation that in many applications, due to branch divergence and/or small block size, several threads of a warp remain idle and they do not utilize all the registers of a register vector. For serving such partially active warps, this technique logically makes hybrid banks from one fast and one slow subbank. By virtue of serving all accesses from fast subbank, this bank acts as a fast bank and slow-bank remains unused. This increases the number of fast banks which improves performance.

Their fourth technique works on the observation that of all registers allocated to a warp, a few registers, especially those with small ID, are accessed much more intensely. Based on it, this technique preferentially maps registers to fast banks. Registers with large ID are mapped to slow banks and these slow banks are used only if register requirement exceeds the capacity of fast banks, thus minimizing the use of slow banks. However, due to increased conflicts at fast banks, this technique does not improve performance. To address this limitation, their fifth technique gives higher issue priority to fast warps (i.e., those which access fast RF banks frequently) over slow warps to maximally increase the progress difference between them. This allows fast warps to begin memory accesses early which reduces memory contention compared to round-robin policy. They show that their techniques enable higher frequency under PV and also improve performance.

Jing et al. [32] compare the impact of PV on SRAM-based and eDRAM-based RF. In an SRAM-based RF, PV manifests as non-uniform latencies in different cells and due to this, overall frequency is determined by the slowest SRAM cell. In eDRAM-based RF, PV leads to variation in retention time of eDRAM cells, but the nominal latency remains uniform across the cells. Thus, to mitigate the impact of PV, the retention period of every register can be recorded and refresh operations can be individually issued to each register. Thus, on using this fine-grain refresh scheme, the eDRAM cell with lowest retention period impacts only the register where it resides and not the entire RF. They further demonstrate that as the severity of PV increases, the energy efficiency of SRAM-based RF degrades much faster than that of eDRAM-based RF and thus, eDRAM-based RF is expected to provide better scalability with ongoing technological scaling.

6 Conclusion and Future Outlook

In this paper, we provided a comprehensive survey of techniques for designing and managing GPU register file. We included techniques related to performance, energy and reliability issues in GPU RF and classified them on important parameters to bring out their similarities and differences. We now discuss some directions that are worthy of further investigation.

To leverage the unique strengths of both CPU and GPU architectures, fused CPU-GPU chips have been designed which integrate them on the same chip [13]. These designs present both challenges and opportunities due to closer CPU-GPU interaction and limited area/power budgets. It will be interesting to see how GPU RF management policies can be adapted to address unique challenges in fused CPU-GPU chips.

It is expected that with increasing demand for throughput, RF size per SM and per chip will also increase. As the amount of chip resources devoted to RF increase, its management will become even more crucial for meeting performance targets and resource constraints. We believe that this issue needs to be simultaneously addressed at different levels of system-stack. At circuit level, novel bank/array organizations and high-density memory technologies can reduce energy/area footprints. At microarchitectural level, techniques such as register sharing, data compression and near-threshold voltage operation, etc. can be integrated to bring the best of them together. At OS and compiler level, mechanisms such as instruction scheduling, changing memory layout and profiling can be used to improve the efficacy of runtime schemes. Finally, insights into application behavior, such as their inherent resilience, can be used to improve reliability and relax safety margins.

References

[1] L. Yu, Y. Pei, T. Chen, and M. Wu, "Architecture supported register stash for GPGPU," *JPDC*, 2015.

[2] NVIDIA, "CUDA C Programming Guide," http://docs.nvidia.com/cuda/cuda-c-programming-guide, 2015.

[3] J. Lim, N. B. Lakshminarayana, H. Kim, W. Song, S. Yalamanchili, and W. Sung, "Power modeling for gpu architectures using mcpat," *ACM TODAES*, vol. 19, no. 3, p. 26, 2014.

[4] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "GPUWattch: enabling energy optimizations in GPGPUs," *ISCA*, pp. 487–498, 2013.

[5] NVIDIA, "NVIDIA's Next Generation CUDA Compute Architecture: Fermi," http://goo.gl/X2AI0b, 2009.

[6] NVIDIA, "NVIDIA's Next Generation CUDA Compute Architecture:Kepler GK110/210 ," http://goo.gl/qOSWW1, 2014.

[7] M. Harris, "5 Things You Should Know About the New Maxwell GPU Architecture," http://goo.gl/8NV82n, 2014.

[8] D. Kanter, "NVIDIAs GT200: Inside a Parallel Processor," http://www.realworldtech.com/gt200/8/, 2008.

[9] "GeForce GTX Titan X Review: Can One GPU Handle 4K?" http://goo.gl/XajvIj, 2015.

[10] Intel, "Intel Itanium Processor 9500 Series Reference Manual," http://goo.gl/xy5m7G, 2012.

[11] W.-k. S. Yu, R. Huang, S. Q. Xu, S.-E. Wang, E. Kan, and G. E. Suh, "SRAM-DRAM hybrid memory with applications to efficient register files in fine-grained multi-threading," in *ISCA*, 2011, pp. 247–258.

[12] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying GPU microarchitecture through microbenchmarking," in *ISPASS*, 2010, pp. 235–246.
[13] S. Mittal and J. Vetter, "A Survey of CPU-GPU Heterogeneous Computing Techniques," *ACM Computing Surveys*, vol. 47, no. 4, pp. 69:1–69:35, 2015.
[14] J. Lucas, S. Lal, M. Andersch, M. Alvarez-Mesa, and B. Juurlink, "How a single chip causes massive power bills GPUSimPow: A GPGPU power simulator," *Intl. Symp. on Performance Analysis of Systems and Software*, pp. 97–106, 2013.
[15] N. Goswami, B. Cao, and T. Li, "Power-performance co-optimization of throughput core architecture using resistive memory," in *HPCA*, 2013, pp. 342–353.
[16] E. Atoofian, "Reducing shift penalty in domain wall memory through register locality," *Intl. Conf. on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, pp. 177–186, 2015.
[17] M. Mao, W. Wen, Y. Zhang, Y. Chen, and H. Li, "Exploration of GPGPU register file architecture using domain-wall-shift-write based racetrack memory," in *DAC*, 2014, pp. 1–6.
[18] C.-C. Hsiao, S.-L. Chu, and C.-C. Hsieh, "An Adaptive Thread Scheduling Mechanism With Low-Power Register File for Mobile GPUs," *IEEE Transactions on Multimedia*, vol. 16, no. 1, pp. 60–67, 2014.
[19] N. Jing, S. Chen, S. Jiang, L. Jiang, C. Li, and X. Liang, "Bank stealing for conflict mitigation in GPGPU Register File," in *ISLPED*, 2015, pp. 55–60.
[20] M. Moeng, H. Xu, R. Melhem, and A. K. Jones, "ContextPreRF: Enhancing the Performance and Energy of GPUs With Nonuniform Register Access," *IEEE Trans. on VLSI*, 2015.
[21] J. Tan, Z. Li, and X. Fu, "Soft-error reliability and power co-optimization for GPGPUS register file using resistive memory," in *Design, Automation & Test in Europe*, 2015, pp. 369–374.
[22] M. Namaki-Shoushtari, A. Rahimi, N. Dutt, P. Gupta, and R. K. Gupta, "ARGO: aging-aware GPGPU register file allocation," in *Intl/ Conf. on Hardware/Software Codesign and System Synthesis*, 2013, p. 30.
[23] S. Mittal and J. S. Vetter, "A Survey of Methods for Analyzing and Improving GPU Energy Efficiency," *ACM Computing Surveys*, vol. 47, no. 2, pp. 19:1–19:23, 2015.
[24] N. B. Lakshminarayana and H. Kim, "Spare register aware prefetching for graph algorithms on GPUs," in *HPCA*, 2014, pp. 614–625.
[25] S. Mittal, "A Survey Of Techniques for Managing and Leveraging Caches in GPUs," *Journal of Circuits, Systems, and Computers*, vol. 23, no. 08, p. 1430002, 2014.
[26] J. Tan and X. Fu, "Mitigating the Susceptibility of GPGPUs Register File to Process Variations," in *International Parallel and Distributed Processing Symposium*, 2015, pp. 969–978.
[27] S. Mittal and J. Vetter, "A Survey of Techniques for Modeling and Improving Reliability of Computing Systems," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2015.
[28] T. L. Falch and A. C. Elster, "Register Caching for Stencil Computations on GPUs," in *Int. Symp. on Symbolic and Numeric Algorithms for Scientific Computing*, 2014, pp. 479–486.
[29] S. Mittal, J. S. Vetter, and D. Li, "A Survey Of Architectural Approaches for Managing Embedded DRAM and Non-volatile On-chip Caches," *IEEE TPDS*, 2015.
[30] S. Mittal, M. Poremba, J. Vetter, and Y. Xie, "Exploring Design Space of 3D NVM and eDRAM Caches Using DESTINY Tool," Oak Ridge National Laboratory, USA, Tech. Rep. ORNL/TM-2014/636, 2014.
[31] J. Vetter *et al.*, "Opportunities for nonvolatile memory systems in extreme-scale high performance computing," *Computing in Science and Engineering (CiSE)*, vol. 17, no. 2, pp. 73 – 82, 2015.
[32] 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," *International Symposium on Computer Architecture*, pp. 344–355, 2013.
[33] Z. Li, J. Tan, and X. Fu, "Hybrid CMOS-TFET based register files for energy-efficient GPGPUs," in *International Symposium on Quality Electronic Design (ISQED)*, 2013, pp. 112–119.
[34] X. Liu, M. Mao, X. Bi, H. Li, and Y. Chen, "An efficient STT-RAM-based register file in GPU architectures," in *ASP-DAC*, 2015, pp. 490–495.
[35] V. Jatala, J. Anantpur, and A. Karkare, "Improving GPU performance through register sharing," *CoRR*, vol. abs/1503.05694, 2015.
[36] H. Jeon and M. Annavaram, "GPGPU Register File Management by Hardware Co-operated Register Reallocation," Univ. of Southern California, Tech. Rep. CENG-2014-05, 2014.
[37] 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, pp. 235–246.
[38] N. Jing, H. Liu, Y. Lu, and X. Liang, "Compiler assisted dynamic register file in GPGPU," in *International Symposium on Low Power Electronics and Design*, 2013, pp. 3–8.
[39] M. Gebhart, S. W. Keckler, and W. J. Dally, "A compile-time managed multi-level register file hierarchy," in *International Symposium on Microarchitecture*, 2011, pp. 465–476.
[40] D. Jeong, Y. Oh, Y. Park, and J. Lee, "An eDRAM-Based Approximate Register File for GPUs," *Design & Test*, 2015.
[41] 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," *Design Automation Conference*, pp. 103:1–103:6, 2015.
[42] J. Wang and Y. Xie, "A Write-Aware STTRAM-Based Register File Architecture for GPGPU," *ACM Journal on Emerging Technologies in Computing Systems*, vol. 12, no. 1, p. 6, 2015.
[43] D. J. Palframan, N. S. Kim, and M. H. Lipasti, "Precision-aware soft error protection for GPUs," *HPCA*, pp. 49–59, 2014.
[44] S. Lee, K. Kim, G. Koo, H. Jeon, W. W. Ro, and M. Annavaram, "Warped-compression: enabling power efficient GPUs through register compression," *ISCA*, pp. 502–514, 2015.
[45] M. Abdel-Majeed and M. Annavaram, "Warped register file: A power efficient register file for GPGPUs," in *Int. Symp. on High Performance Computer Architecture*, 2013, pp. 412–423.
[46] P. Xiang, Y. Yang, M. Mantor, N. Rubin, L. R. Hsu, and H. Zhou, "Exploiting uniform vector instructions for GPGPU performance, energy efficiency, and opportunistic reliability enhancement," in *ICS*, 2013, pp. 433–442.
[47] S.-L. Chu, C.-C. Hsiao, and C.-C. Hsieh, "An energy-efficient unified register file for mobile GPUs," in *Intl. Conf. on Embedded and Ubiquitous Computing (EUC)*, 2011, pp. 166–173.
[48] M. Gebhart, S. W. Keckler, B. Khailany, R. Krashinsky, and W. J. Dally, "Unifying primary cache, scratch, and register file memories in a throughput processor," *MICRO*, 2012.
[49] S. Z. Gilani, N. S. Kim, and M. J. Schulte, "Power-efficient computing for compute-intensive GPGPU applications," in *HPCA*, 2013, pp. 330–341.
[50] S. Mittal, "A survey of architectural techniques for improving cache power efficiency," *Elsevier Sustainable Computing: Informatics and Systems*, vol. 4, no. 1, pp. 33–43, March 2014.
[51] G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Base-delta-immediate compression: practical data compression for on-chip caches," in *PACT*, 2012, pp. 377–388.
[52] S. Mittal, "A Survey of Techniques for Approximate Computing," *ACM Computing Surveys*, 2016.
[53] J. Tan, Y. Yi, F. Shen, and X. Fu, "Modeling and characterizing GPGPU reliability in the presence of soft errors," *Parallel Computing*, vol. 39, no. 9, pp. 520–532, 2013.
[54] N. Farazmand, R. Ubal, and D. Kaeli, "Statistical fault injection-based AVF analysis of a GPU architecture," *SELSE*, 2012.
[55] S. Mittal, "A Survey Of Architectural Techniques for Managing Process Variation," *ACM Computing Surveys*, 2016.

Sparsh Mittal received the B.Tech. degree in electronics and communications engineering from IIT, Roorkee, India and the Ph.D. degree in computer engineering from Iowa State University, USA. He is currently working as a Post-Doctoral Research Associate at ORNL. His research interests include non-volatile memory, memory system power efficiency, cache and GPU architectures.