Cache-Emulated Register File: An Integrated On-Chip Memory Architecture for High Performance GPGPUs

Naifeng Jing, Jianfei Wang, Fengfeng Fan, Wenkang Yu, Li Jiang, Chao Li, Xiaoyao Liang*
Advanced Computer Architecture Laboratory
Shanghai Jiao Tong University, Shanghai, China, 200240

***Abstract*—The on-chip memory design is critical to the GPGPU performance because it serves between the massive threads and the huge external memory as a low-latency and high-throughput data communication point. However, the existing on-chip memory hierarchy is inherited from the conventional CPU architecture and is oftentimes sub-optimal to the SIMT (single instruction, multiple threads) execution. In this study, we surpass the traditional memory hierarchy design and reform the on-chip memory into an integrated architecture with the cache-emulated register file (RF) capability tailored for high performance GPGPU computing. With the lightweight support from ISA, compiler and the modified microarchitecture, this integrated architecture can dynamically emulate a variable-sized RF and a cache in a uniform way. Evaluation results demonstrate that this novel architecture can deliver better performance and energy efficiency with smaller on-chip memory size. For example, it can gain an average of 50% performance improvement for the cache-sensitive applications.**

I. Introduction

In the last couple of years, GPGPU has emerged as a pervasive computing engine for many general-purpose applications. The superior performance achieved largely owes to its high computing throughput from the massive threads and the large memory bandwidth from the external memory subsystem. As a bridge between them, the low latency data communication is realized through a complex on-chip memory hierarchy, which generally leverages diversified memory resources such as RF, shared memory, the first-level data (L1D) cache as well as the constant and texture caches. The major function of RF is to quickly deliver operands to the parallel threads, and the L1D cache is designed to reuse the local data without paying the expensive round-trip cost to the main memory.

As GPGPU is oriented for throughput, the on-chip memory capacity can be critical to the performance. In presence of the massive threads, RF is usually designed quite large such as 128KB per streaming multiprocessor (SM) in Fermi architecture [1], which is twice as large as the total capacity of the shared memory and L1D. In Kepler [2], the RF has been doubled in capacity for even higher parallelism.

As modern GPGPU is becoming area- and power-limited [3], simply adding on-chip memory resources for performance improvement is no longer economic. On the other hand, the on-chip memory is significantly under-utilized in the contemporary GPGPUs as we will show in the later sections. This is because the existing design choice mainly follows what has been done in CPUs such as separating the RF and cache into two components and optimizing them individually [4–11], missing a holistic view of the on-chip memory functionality in GPGPUs. Gebhart et.al [3] was among the first to shed light on this issue by considering a unified on-chip memory structure and achieved significant performance improvement. However, their work still divided the unified memory into RF and L1D statically before a kernel launch without considering the dynamic and interactive behavior between them. In another word, although their work physically organized the RF and L1D into a unified memory component, logically it did not deviate too much from the traditional CPU view of the memory architecture, and cannot make full use of the precious on-chip resources as we identified in this study.

The SIMT execution poses challenges but also new opportunities for a revolutionized on-chip memory architecture. By carefully studying the tradeoffs in the contemporary design, we propose a novel integrated architecture using monolithic on-chip memory with cache-emulated RF capability. This means that the entire memory physically works just like a traditional cache. However, by some lightweight modifications on the ISA, compiler and the microarchitecture, this cache can effectively emulate all the RF functionality and break the distinction of cache or RF during the entire kernel execution. The novel architecture neatly fits the natural characteristics of GPGPU behavior, leading to greater efficiency with even smaller on-chip memory size.

The rest of the paper is organized as follows: Section 2 introduces the baseline GPGPU on-chip memory architecture and the motivation. Section 3 introduces the proposed detailed design. Section 4 and 5 give the experimental methodology and results. Section 6 reviews the related work. Finally, section 7 concludes the paper.

II. Motivations

A. Reviewing On-Chip Memory

The on-chip memory in GPGPUs mainly consists of RF, L1D cache and shared memory. Other types of memories are also available, such as the constant and texture caches but are not discussed in this paper since they are not essential for general-purpose computing.

978-1-5090-3508-3/16/$31.00

The RF is generally large to support the zero-cost warp switching. To provide a high bandwidth without excessive complexity, it leverages a multi-banked structure for the simultaneous reads and writes in the SIMT execution. To minimize the bank conflicts, registers are interleaved across banks such that registers from different banks can be simultaneously accessed. This RF organization allows the interconnects between the banks and lanes to be reasonably designed with power efficiency, and has been used in many prior studies [3–5, 12].

The L1D cache is likely to be a classic set-associative structure as exposed in [13, 14] by their extensive micro-benchmarking on real nVidia GPU cards. They found that the L1D cache uses 128B block size with a 32-set, 4-way structure for the default 16KB configuration. The cache can be reconfigured as 48KB in capacity.

In terms of the memory size, the RF capacity is much larger than L1D and the gap tends to grow in the future. According to nVidia's documents [1, 2], the shared memory and L1D are as fast as RF in the access speed.

B. Under-Utilization in RF

RF is the biggest on-chip memory resource. However, due to the varying parallelism in different applications, the registers may be over provisioned generating inactive registers, namely RF *holes* in this paper.

Static holes due to limited TLP. Given the large number of registers, it is possible that some kernels cannot use them up due to the insufficient parallelism. Because the RF utilization is a function of the number of registers per thread and the total number of concurrent threads, it is rare that a kernel can make an exact usage of all the available registers without any hole. RF typically experiences static under-utilization. This phenomenon is widely acknowledged and can be leveraged for power optimization [5, 15–17].

Dynamic holes due to register lifetime. It is well known that the allocated registers are not persistently active during the execution accounting for the dynamic holes in RF. The register occupancy is constantly changing with only a short period reaching the peak. The dynamic holes in a single thread can be identified by the register liveness analysis at the compilation time. However, the total number of dynamic RF holes is unknown beforehand due to the independent warp execution scheduled by hardware. As a result, the compiler has to reserve the peak register usage for each warp by assuming that all the warps are running in line, which conservatively leads to the register over-provisioning.

Evaluation on holes. To learn the statistics of the static and dynamic holes, we evaluate them in Fig. 1. Static holes are represented by the bar stacked on the top, which are available for the entire kernel execution. Dynamic holes are represented by the bar laid in the middle. As their numbers vary cycle by cycle, we calculate an averaged number throughout the execution. The active registers are represented by the bottom bar, which are also averaged on a cycle basis.

As the RF capacity is generally large in GPGPUs, there are many *non-register-limited* kernels whose TLP is limited by thread concurrency and other resources. Averagely, the static and dynamic holes account for over 50% of the entire RF, implying a non-ignorable resource redundance. Similar results have been observed in [3, 15] on static holes and [5, 16–18] on both. Note that the wasted RF resource even exceeds the size of L1D cache.

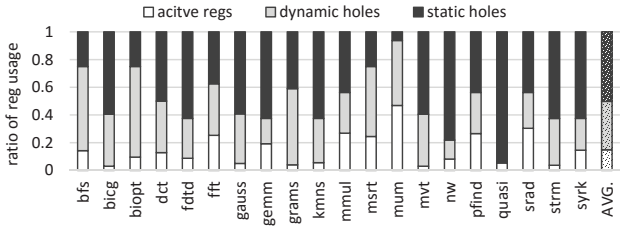

Fig. 1: Breakdown of different register usages.

C. Thrashing in L1D

Recently, the L1D thrashing problem in GPGPUs has drawn great attention. It essentially attributes to the small cache capacity compared to the large working data set. For example, the 16KB L1D cache may be shared by 1,536 threads, which means only a few bytes are reserved for each thread. The thread contention then causes frequent premature evictions of cache lines.

Generally, there are two approaches for the thrashing problem: 1) design a cache with larger capacity or associativity; 2) design a cache more smartly to reduce the early evictions of useful data. As far as we know, contemporary studies mainly focus on the latter, centralizing on the *cache bypassing* with thread throttling [7, 8], scheduling [10], request reordering [6] or prioritization [9]. However, due to the dynamic and irregular memory behavior in the emerging GPGPU applications, it may be difficult to find a universally effective technique in the second category. In contrast, the first approach is relatively simple to improve the performance, but was rarely exploited so far due to the difficulty of adding costly caching hardware. As we have observed the significant under-utilization in RF, it actually offers a unique opportunity as a complementary memory resource for the L1D cache. The only problem left is how to manage the memory resource with intelligence.

D. Motivation for an Integrated Architecture

The hierarchical on-chip memory design has been a time-honored approach for high performance facing the growing speed disparity between the inside logics and outside memory. In the CPU regime, it is proved to be efficient with a small number of registers offering immediate accesses, and a large cache for better data locality. However, prior study [4] has shown that it is not always the case in GPGPUs. Fig. 2(a) demonstrates the situation on the number of register reuse (reads) after definition (writes). From this plot, we see that around 60% registers are used less than twice, and less than 10% are used more than twice. This largely owes to the stream processing nature commonly in GPGPU applications, which leads to relatively little register reuse.

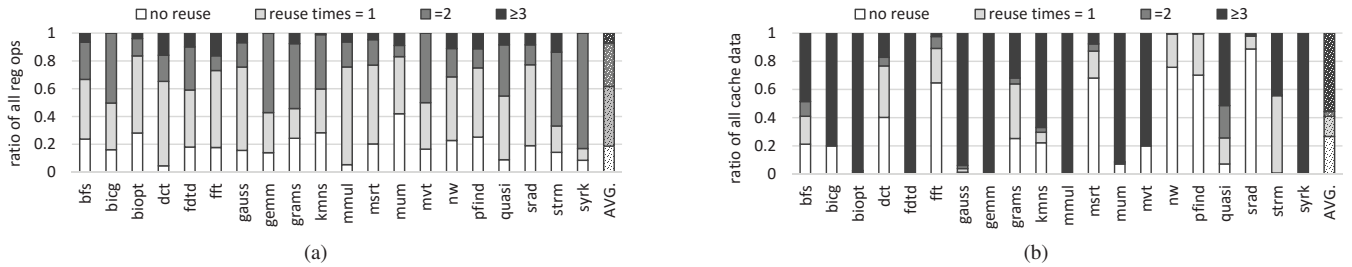

Fig. 2: The ratio of (a) register reuse count, (b) cache reuse count.

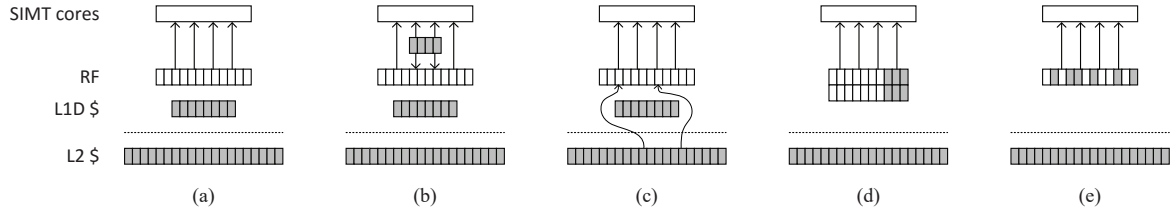

Fig. 3: Comparison of different on-chip memory designs.

We also study the L1D cache behavior to learn their reuse patterns by capturing the accesses to the same address in GPGPU applications. The results are shown in Fig. 2(b), where we find that the cache reuse varies from once to tens of times. It is important to note that on average, nearly 60% cache operands are reused over three times. We also observe that the memory accesses may account for a great ratio of the total on-chip data accesses, for example, around 30% as normalized in applications like "bfs", "quasi" and "syrk" (not shown in the plots). From the locality point of view, these frequently reused cache operands should be kept on-chip like registers for fast accesses. However, dynamic cache addresses are generally hard to predict at the compilation time and the very limited cache capacity may lead to excessive conflicts and early evictions.

In this paper, we propose an integrated on-chip memory architecture to solve the dilemma. It leverages the under-utilized RF to accommodate the L1D data for better locality. It integrates the on-chip data accesses at runtime by breaking the distinction between RF and cache for resource sharing, and applies a uniform management to leverage the vacant registers for the caching functionality dynamically. In other words, the RF holes can constantly (for static holes) or opportunistically (for dynamic holes) serve the cache content without expanding the actual storage capacity. In this sense, the proposed architecture not only physically places the data together, but also complements them seamlessly with a smart policy as introduced later in this paper.

III. Proposed Architecture

In this section, we will propose an integrated on-chip memory architecture with cache-emulated RF capability.

A. Design Overview and Challenges

Traditional GPGPUs separate RF and L1D cache as shown in Fig. 3(a). For a better on-chip memory utilization, there are other alternatives proposed in recent literatures. For instance, RF caching [4] in Fig. 3(b) introduces a small but fully-associative register cache in front of the main RF. It improves the energy efficiency but at the cost of another memory level which complicates the overall design. Cache bypassing [6–8] in Fig. 3(c) tries to mitigate L1D thrashing by differentiating cache accesses. However, it ignores the role of RF that is equally important. Another work proposes a unified memory [3] in Fig. 3(d), where the unified memory can be repartitioned statically at each kernel launch. However, the coarse-grained approach may fail to recognize the imbalanced resource utilization during the kernel execution. Once partitioned, the unified memory is logically viewed as the separate RF and cache, requiring the hardware support for addressing and accessing modes that complicates the memory implementation. Our proposed integrated architecture is illustrated in Fig. 3(e). Compared with Fig. 3(d), there are two apparent distinctions. Firstly, the integrated memory is no longer partitioned and always treated as a whole, so that it can capture the dynamic imbalance between RF and cache during the kernel execution. Secondly, the memory is physically designed as a cache so that the hardware implementation is much simplified. Our proposed integrated architecture is also complementary to Fig. 3(c) by optimizing the cache thrashing with virtually up-sized cache and hence putting less stress on cache bypassing.

To achieve the integrated monolithic on-chip memory, three design challenges must be addressed:

- The first challenge is how to integrate RF and cache into the monolithic memory. We propose to use the classic set-associative cache structure for the cache-emulated RF capability, and the traditional on-chip memory hierarchy can be simplified.
- The second challenge is how to manage the data in the memory. Conventionally, RF and cache are addressed differently. We propose a register re-indexing scheme to

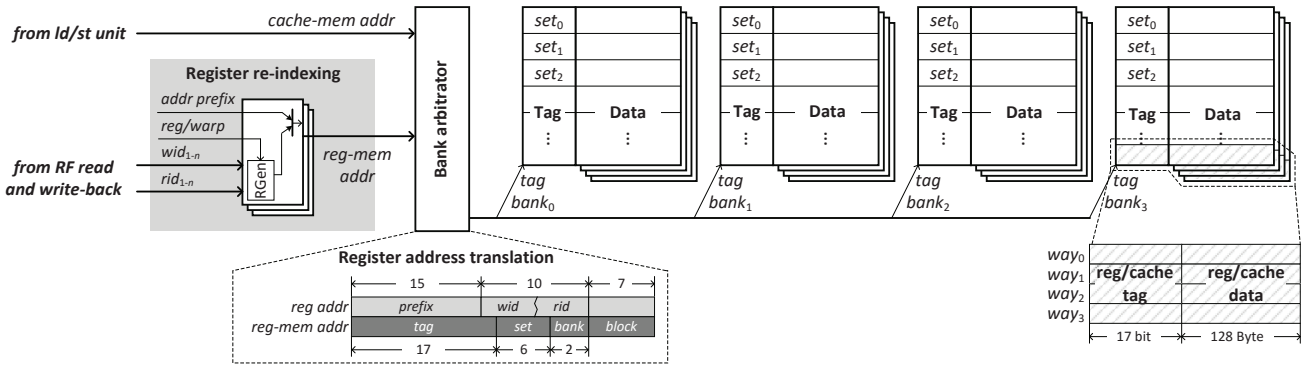

Fig. 4: The integrated on-chip memory architecture.

manage the cache-emulated RF uniformly because the memory is now physically designed as a cache.

- The third challenge is how to design a replacement policy to identify the most frequently used data in the program. Traditionally, registers are explicitly kept for highly frequent reuse. In the integrated architecture, we also need to identify the most frequently used data and design a proper policy to achieve better data locality.

B. Integration as a Monolithic Memory

As illustrated in Fig. 4, the monolithic memory can be fairly simple by applying a cache alike structure. It lies between the execution units and L2 cache, mainly consisting of an integrated data array and a tag array.

Integrated data array. The integrated data array holds the register and cache data simultaneously. On one hand, it behaves like RF by serving operands to the execution units and accepting the write-back values. On the other hand, it works as the L1D communicating with the L2 via the MSHR (miss status holding registers) unit on missed memory requests.

As there is no noticeable speed disparity between RF and L1D in modern GPGPUs [1–3], the integrated data array applies the same multi-banked structure as the traditional RF. This ensures the memory primitives, such as access time, port number, bandwidth, area and power remain unchanged in the proposed design. For example, the machine cycle time is not affected in the integrated data array by using the same size and ports as in the traditional RF. Note that with a better resource utilization, a 128KB integrated data array can still deliver satisfactory performance improvement over the traditional separate 128KB RF and 16KB L1D.

Tag array. To match the tag lookup bandwidth from multiple simultaneous instructions, we design the tag array using a banked scheme similar to the data array by distributing the requests onto multiple tag banks for parallel lookups. Assume the data array of the 128KB RF per SM is made up of 32 banks with 4KB registers per bank and each bank is of 16B wide that accommodates 4 registers from 4 consecutive threads [3–5]. Therefore, up to 8 banks are accessed for a single register from all 32 threads in a warp. The tag array can be accordingly partitioned into 4 banks, and each tag bank governs 8 data banks to offer the same lookup rate.

We also insert a "tagging" stage into the pipeline between decoding and RF accessing, during which the register re-indexing (explained later) and tag lookup for both reads and writes can be performed. There are two reasons for adding this stage. The first is for better timing since re-indexing and lookup incur extra delay on register accessing. Adding the new stage balances the cycle time. The second reason is to save power because the 1,024-bit GPGPU data is much wider than CPU. It is more power economic to perform a tag lookup first for a confirmed hit before the real data is read out.

In the banked structure, conflicting requests on the same bank have to be serialized and may potentially cause pipeline stalls. Given the fact that the conflicts are also common in the traditional RF, the buffering units have already been built to tolerant the variable access latencies. Therefore, we can reuse the existing units and pipeline control logic in our architecture. With most recent architectures tend to mitigate the conflicts with static information, we can leverage the same expertise to coordinate our proposed integrated memory on the conflict problem. Note that there will be no conflicts in the data array because we always perform the tag lookup first and grant the data access in the next cycle.

C. Uniform Memory Management

The second challenge is to manage the register and cache accesses in a uniform way. As the integrated memory is designed just like a typical L1D cache, the same ld/st unit in the baseline can be used and the cache accesses to this memory can be trivial.

To enable register access to this classic cache structure, the register identifier need to be translated into memory addresses to locate operands in the integrated memory. In this study, we propose a simple register re-indexing that augments the original register ID to form a new "reg-mem" address. As shown in Fig. 4, the re-indexing first cascades the register ID and the corresponding warp ID to form a register address by a "RGen" unit as in the traditional RF. The total 10-bit register address (for 1,024 warp registers) is then left-shifted by 7 bits accounting for the 128B block size and augmented with a 15-

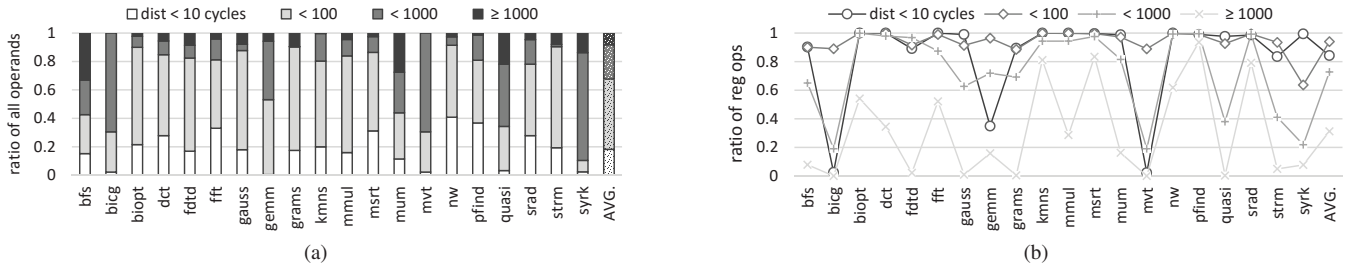

Fig. 5: (a) Reuse distance values of all the operands. (b) The ratio of register operands.

bit prefix to form the final reg-mem address. On interpreting this address, assuming we have 256 cache sets that are evenly distributed into 4 tag banks, the lowest 2 bits next to the block offset select which tag bank to access, and the next 6 bits index to the set location in the bank. All the remaining bits are the true tag stored in the tag array.

This indexing scheme is designed with several considerations. Firstly, it is fair for both register and cache data by distributing them across all the sets. Intuitively, a register with long reuse interval (between last read and re-definition) is perfect for holding the cache data. In our scheme, since registers are associatively placed into a set, even a register with short reuse interval can also be fully leveraged, offering more flexibility for cache data. This is a significant advantage over the traditional RF architecture. Secondly, sequential registers are distributed to sequential banks. This helps to reduce the bank conflicts for parallel register accesses. At the same time, cache data with spatial locality are also spread to different banks for simultaneous accesses since we keep the original linear cache addressing mode.

To avoid interfering with the real memory addresses, the reg-mem addresses can be designated to an isolated space in the global memory via address prefixing. Different prefixes can be applied in different SMs to form the reg-mem addresses such that the entire RF can be one-to-one mapped into the global memory space. We adopt this simple scheme because the global memory is generally thousands of times larger than the RF size. We believe that the impact on the total memory space can be negligible.

D. Replacement Policy

The third challenge is to identify the most frequently reused data and maintain them in the integrated memory for better locality. We find that a simple LRU based replacement supplemented by an optional register prioritization scheme can well meet the need. To be specific, we investigate two policies: *LRU-fair* and *LRU-prioritized.*

1) Design considerations: The LRU policy generally works well for the integrated architecture because it is good at identifying the frequently reused data with short reuse distances. Fig. 5(a) confirms this by collecting the dynamic reuse distances for all the register and cache data using the traditional architecture. We see that the data with short reuse distances accounts for a large portion. For example, around 60% data are reused within 100 cycles, while around 90% data are reused within 1000 cycles. Accordingly, as shown in Fig. 5(b), around 70% of those reused data within 1000 cycles are register operands. This implies that the LRU policy can well keep the frequently reused register operands and filter out those rarely reused. The filtered entries can be consequently occupied by the frequently reused cache data, satisfying the needs for both RF and L1D cache.

For the LRU-fair policy, the integrated memory applies the exact LRU-based line replacement as in a typical cache. When new data arrives, it selects a cache line to replace according to the aging information, regardless of what kind of data (register or cache) in this entry. In other words, as a fair policy, register and cache data are equal and mutually replaceable within a set. This mechanism is simple and faithful to cache, but may occasionally cause register spilling to the secondary memory, because the memory is no longer aware of what kind of data in it. Fortunately, register spilling is not new to GPGPU hardware because spilling to the lower level memory is already supported when the number of registers per thread is insufficient. We can reuse the datapath and logic behind the special ld/st instructions from/to the local memory via the translated reg-mem addresses (actually in the global memory space) to fulfill the spilling operations required by LRU-fair.

When LRU-prioritized is applied, the replacement is slightly modified by checking the associated data state. As explained later, the state can be “register” or “cache”, which is then leveraged to prioritize register over cache data using a prioritized replacement. The newly arrived register can replace cache data if no other inactive entries are found, while the state protects the register data from being replaced by cache data. In this case, there will be no register spilling because all the registers are re-indexed for a full mapping into the integrated memory. This implies that the cache managed integrated memory can faithfully emulate the RF with no spills.

Intuitively, LRU-prioritized better protects the register data from spilling to the secondary memory. However, it may reduce the performance advantage by blocking useful cache data from being stored into the integrated memory if all the ways in the target set are occupied by registers. In contrast, LRU-fair makes a flexible use of the integrated memory but may hurt the performance due to register spilling. In a nutshell, it is hard to judge the two schemes, and we will investigate them in the experiments. Another extreme of the spectrum

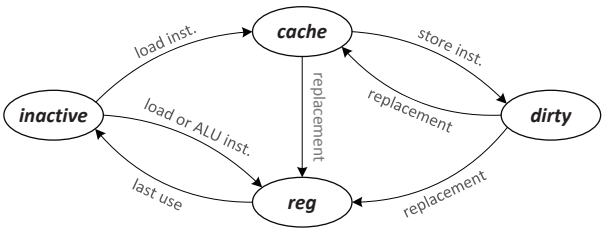

Fig. 6: Memory state transitions (LRU-prioritized).

	Baseline (separate)	Proposed (integrated)
# SMs	15	
warp size	32 threads	
scheduler	2 GTO warp schedulers per SM	
TLP	1536 threads, 48 warps, 8 CTAs per SM	
on-chip memory	RF: 128KB, 32-bank L1D: 16KB, 4-way tag: ≈0.3KB, 4-way	128KB, 4-way, 32-bank n/a ≈2KB, 4-way
	128B cache line, LRU 48KB shared memory	
L2 cache	unified, 64KBx12, 128 line size, 8-way, LRU	

TABLE I: Architecture configurations in simulator.

is to prioritize cache data, but the performance can be poor as expected because register latency is more critical. We believe that with the current compilation framework following a separate RF and cache model, there can be continuous efforts on the replacement policy with different hardware constraints.

2) ISA and Compiler Support for LRU-Prioritized: To identify the RF holes explicitly for holding cache data, we leverage lightweight ISA and compiler support. We also label the memory entries with different states. Note that these enhancements are only required by the LRU-prioritized policy. The LRU-fair policy does not require these. Instead, LRU-fair exploits the dynamic holes by aging information to retrieve the dead registers for cache sharing, although not as timely as the LRU-prioritized.

ISA and compiler assistance. To identify the RF holes, we leverage the register liveness analysis that has already been a classic technique in the modern compiler. The analysis tells the definition-use chain of each register. A register becomes live after it is defined and dies after its last use. The liveness analysis can be combined with static data-flow analysis, such that the registers across the conditional branches and predictive instructions with one or more predecessors and successors can be properly analyzed [19]. We conservatively declare a warp register to death only when all the 32 threads reach the same post-dominant instruction that reads it for the last time.

The analyzed liveness information can be encoded through ISA enhancement by attaching a bit to the instruction to define the register state at runtime. This bit is initiated by the compiler. According to the PTX standard [20], each instruction has up to three source registers with a semantic of `op dest,sa,sb,sc`. As the register indexed by `dest` always starts a new definition of an active register, we can extend the instruction encoding by adding 3 bits to indicate the liveness of `sa,sb,sc`, respectively. The annotated instructions pass along the pipeline and the hardware captures the register liveness information at runtime.

The proposed solution has low cost and can identify the static and dynamic holes at the same time. There are other complementary alternatives as proposed in [16, 17], such as using direct commands to invalidate dead registers, which can also be adopted in our study.

State transitions. There are 4 states for each entry in the integrated memory, namely *inactive*, *reg*, *cache* and *dirty*, implying the associated entry is vacant, holding register, clean cache or dirty cache data. Fig. 6 gives the state transition diagram for a single entry.

Transition *inactive* $\rightarrow$ *reg* typically happens on a register writing in an ALU instruction, setting up a new register entry in the memory. The backward transition happens when the register is declared dead by the encoded liveness bit. After that, the entry is retrieved for another register or cache data. Transition *inactive* $\rightarrow$ *cache* happens on a load instruction. The *cache* state stays until the associated entry is replaced by a register writing with state transition to *reg*, or being updated by a store instruction with state transition to *dirty*. Note that the dirty data should be written back to L2 if another register or cache data is going to occupy the entry.

Each load instruction takes two actions. First, it sends the fetched data and caches it in the memory. Then, it updates the destination register with the fetched data. In most cases, the locations of the registered and cached data will be different yielding two copies of the same data in the memory. No extra work is done for the load instruction compared to the traditional design because it needs to write the RF and L1D cache anyway.

E. Design Cost Analysis

The major functional block of the integrated memory is the data array, which is sized and banked in the same way as the RF in the traditional GPGPUs. By default, it is set to 128KB, and partitioned into 256 sets if we apply a 4-way set-associative structure. Therefore, the tag array requires 2.125KB storage assuming a 32-bit memory address. Each entry requires 2 bits for the 4 states in LRU-prioritized. In total, our proposed architecture requires 130.375KB storage versus about 144KB storage in the traditional architecture (128KB RF + 16KB L1D + 0.3KB tag/state). We need some extra logics for the register re-indexing and state transition. The re-indexing only cascades and shifts address components and the state transition is a very simple state machine. The instruction decoder needs to be augmented on the register liveness decoding. However, we believe that the added overhead can be acceptable. The instruction length may not be a problem because the 64-bit length of most SASS instructions as revealed in [14] has reserved sufficient bits for the future ISA extension. The pipeline control for the variable latency and register spilling operations is already built in the traditional GPGPUs. So we can reuse most of the logic without significantly changing the pipeline complexity.

Name	Abbr.	$ sensitivity	Name	Abbr.	$	Name	Abbr.	$
binomialOptions* [21]	biopt	low	bicg [22]	bicg	high	BFS [23]	bfs	high
gramschmidt [22]	grams	low	fdtd2d [22]	fdtd	low	fft [21]	fft	low
matrixMul [21]	mmul	low	gaussian [23]	gauss	low	dct8x8 [21]	dct	low
mergeSort [21]	msrt	high	kmeans [23]	kmns	low	gemm [22]	gemm	high
pathfinder [23]	pfind	low	MUM * [21]	mum	high	mvt [22]	mvt	high
quasirandomGenerator [21]	quasi	high	srad [23]	srad	high	nw [23]	nw	low
streamcluster [23]	strm	high	syrk [22]	syrk	high			

TABLE II: Benchmarks (* marked are register-limited applications).

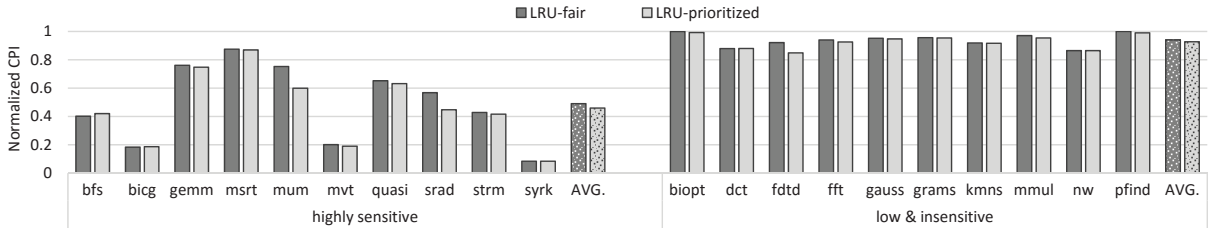

Fig. 7: Normalized CPI using two replacement policies (lower is better).

	E_{read} (pJ/bank)	E_{write} (pJ/bank)	$P_{leakage}$ (mW/bank)
128KB RF	4.62	5.97	1.78
16KB L1D	0.72	0.61	0.19
L1D tag	0.47	0.64	0.20
128KB integrated	the same to 128KB RF		
integrated tag	0.75	0.78	0.32
L2 data	5.52	3.95	2.66
L2 tag	1.05	1.91	0.80

TABLE III: Power primitives.

IV. Evaluation Methodology

For the performance measurement, we use the most up-to-date cycle level architecture simulator GPGPU-Sim v3.2.2 [24] and configure it as Table I. We adopt the narrow RF-banking structure as most prior studies [3–5, 12]. However, our technique is not limited to this specific banking and can be applied to other banking schemes as well. We evaluate the proposed integrated memory of 128KB data array serving as both RF and L1D. It uses the same physical RF design as the baseline for timing compatibility. Additional blocks are dedicated in the added "tagging" pipeline stage, which can sufficiently absorb the delay overhead because the tag lookup is much faster than the data access. Meanwhile, the data access bandwidth in the integrated memory is set equal to the RF bandwidth in the baseline.

Table II shows the 20 benchmarks used in our evaluation from a wide range of applications in CUDA SDK [21], Rodinia [23] and Polybench [22] suites. We use the largest workload if possible to demonstrate the results. These benchmarks can be categorized into two groups of different sensitivities to hardware caching, and are chosen using the baseline configuration according to their sensitivities across the suites. Because GPGPUs are designed for massive threads and the RF is generally large for many applications, we find only two register-limited applications.

Due to the lack of open-source compiler to generate codes with register liveness information, we intercept the compiled kernel before running the simulation at the SASS level, which carries accurate register allocations and other optimizations from the commercial "*nvcc*" compiler. The interception binds our liveness annotation to each register. The simulator is also modified accordingly to accept the annotation to enable the state transition and identify the RF holes at runtime.

For energy evaluation, we use CACTI 6.5 [25] to model different memory structures in the baseline and proposed design. The power primitives are shown in Table III. We collect RF and L1D cache statistics from GPGPU-Sim with these primitives to compute the power in baseline. The power calculation can be similarly done for the integrated architecture by considering the tag lookup, data accessing and all the associated logics. It is worthwhile to point out that the power savings reported in our evaluation could be conservative, because we do not consider the reduced memory requests from the L2 cache to main memory, given the fact that the off-chip memory accesses can be more power hungry.

V. Experimental Results

A. Performance Evaluation

Performance with different replacement policies. We first study the performance with the two replacement policies: LRU-fair and LRU-prioritized. Fig. 7 reports the normalized CPI of the proposed 128KB integrated memory against the baseline with a separate 16KB L1D and a 128KB RF. We see that the two policies deliver almost the same performance most of the time, winning over the baseline performance by an average of 30%. For the cache-sensitive applications, the LRU-prioritized slightly wins on average. The reason has been explained in Section III-D because it faithfully emulates the RF without spilling and timely retrieves the inactive registers for cache use. However, neither of the two policies is constantly better than the other. The results again confirm the necessity to integrate the RF and L1D for a better resource

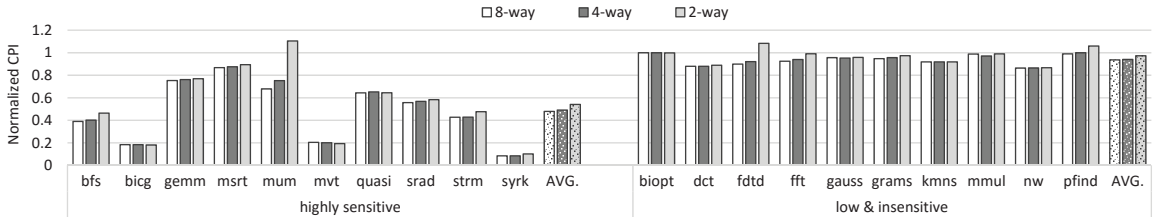

Fig. 8: Normalized CPI using varying associativity (lower is better).

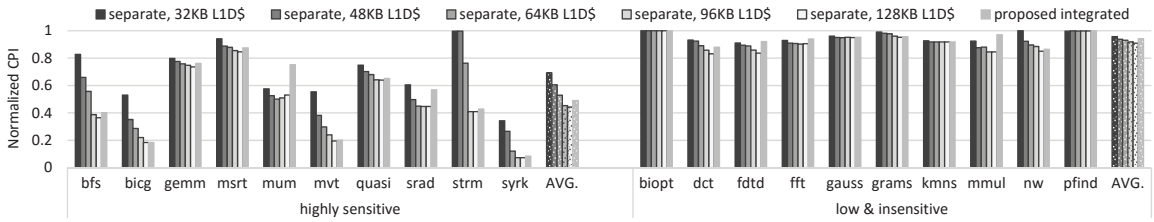

Fig. 9: Performance impact of L1D cache capacity (lower is better).

utilization. Due to their comparable performance, we will only use the LRU-fair to demonstrate the results hereafter.

Performance with associativity. Our proposal enables a variable sized cache because the register and cache data share the same memory during the runtime. Given the fixed capacity, higher associativity can reduce the conflicts in a set. To study the impact, we evaluate the 8-way, 4-way and 2-way designs.

Fig. 8 reports the performance normalized to the baseline by varying the associativity. From the plot, we see that higher associativity usually generates better performance, but the benefit diminishes beyond 8-way in most cases. This is mostly due to the high associativity offers a better co-existence of cache and register data so that the locality of both types can be improved. We find that the 4-way design generally delivers a good performance improvement of around 30% over the baseline with an acceptable hardware complexity. As a result, we will use it to demonstrate all the results hereafter.

Comparison to baselines with different on-chip memory sizes. Since modern GPGPUs can reconfigure the L1D size according to applications, we will compare our performance to the baseline with the varying L1D sizes from 16KB to 128KB as shown in Fig. 9. We observe that for applications that are highly sensitive to caching, the performance gains can achieve up to 2X comparing to a separate 16KB L1D baseline. This is because our integrated architecture virtually increases the L1D capacity. It also outperforms the separate 48KB L1D, which is the alternative configuration in Fermi. In many cases, the performance can be better than a separate 64KB L1D. Owing to the large amount of RF holes identified for cache sharing, our proposed architecture successfully emulates a flexible RF and meanwhile manages a virtually up-sized L1D.

We also evaluate different RF capacity, such as 64KB and 256KB in the baseline with a 16KB L1D. Then, the size of the integrated memory is changed accordingly. In brief, the evaluation results stay almost the same, and we will skip the details for the limited space here.

Comparison to the unified memory structure. The dynamic resource sharing nature distinguishes our architecture from the unified memory structure in [3] that statically divides the RF and cache before a kernel launch. Fig. 10 plots the performance comparison. Due to the unavailability of their in-house benchmarks, we implemented the unified memory with a total size of 128KB that is the same to our design and test the scheme on the same benchmarks in this paper. For the common benchmarks we both used (“bfs”, “dct”, “mum” and “mmul”), the performance in our experiments generally matches with their results in [3], confirming the fidelity of our implementation.

From the plot, we find that our proposed architecture outperforms the unified memory by 25% on average for cache-sensitive applications. The improvement mainly comes from the dynamic RF holes that we can aggressively leverage for cache sharing. Taking “bfs” as an example, Fig. 1 shows that about 25% static holes can be used in the unified memory scheme leading to around 15% performance improvement [3]. With extra 60% dynamic holes in our architecture, we achieve another 40% performance improvement. Similar observations can be found in “srad” and “strm”. For register-limited benchmarks like “mum”, there are very few static holes for the unified memory, and therefore their performance stays almost the same as the baseline. In contrast, our architecture can effectively leverage the 40% dynamic holes and improve the performance by nearly 30% over the baseline. For “quasi” on the opposite, the small number of dynamic holes observed in Fig. 1 leads to a similar performance compared to the unified memory scheme. The benefit of our scheme is that even with no dynamic holes, we can still make full use of the static holes as in the unified memory. Since it is rare to find applications with neither static nor dynamic holes, our architecture can always find the opportunity for resource sharing.

Note that both techniques are less effective for the cache-insensitive applications, but our proposal can provide slightly

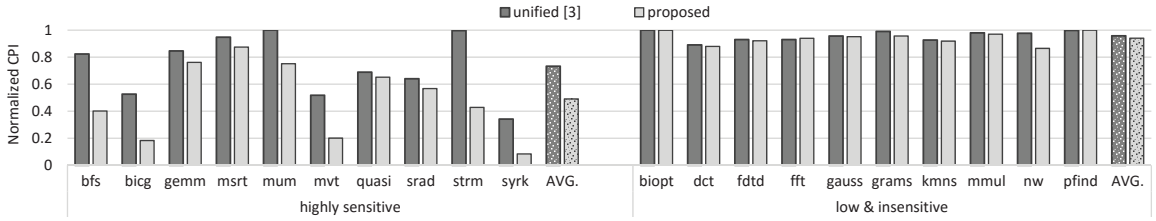

Fig. 10: Performance comparison of the proposed design to the unified memory [3] (lower is better).

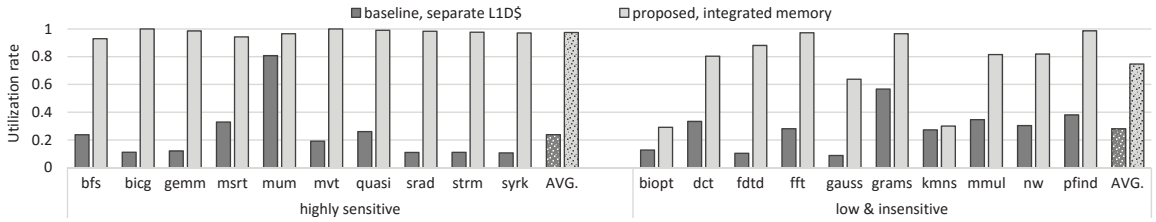

Fig. 11: The on-chip memory utilization rate in the baseline and the proposed design.

better performance owing to the dynamic hole sharing. For those applications, both techniques are complementary to other solutions such as cache bypassing [6, 7], which we believe can open up new opportunities for performance improvement.

B. Detailed Memory Statistics

This section will study the detailed memory statistics for a better understanding of our proposed architecture.

On-chip memory utilization. A higher utilization implies a better usage of the on-chip memory resource. The utilization is calculated as the ratio of the active entries in the integrated memory. In the baseline, it is calculated as the ratio of the active registers plus the number of valid cache lines over the total storage size.

Fig. 11 compares the utilization of the baseline and the proposed architecture, where we observe a significantly increased utilization rate for all the benchmarks. On average, our scheme improves the utilization rate by over 50%. For many applications, the utilization rate can reach almost 100% through dynamic data sharing in the integrated memory. We also notice that "biopt" still suffers a small utilization rate. This is mostly because its working set is very small and the 128KB memory capacity is over-provisioned.

On-chip memory hit rate. The performance gain mostly owes to the greatly increased hit rate in the integrated memory. Fig. 12 shows the hit rate for all the L1D cache accesses compared with the baseline, where we observe a significant hit rate increase over 20% for most cases. For "dct" with a relatively low hit rate, it is worth mentioning that the proposed architecture still increases it from 0.6% to 13%, around 20 times hit rate increase and in turn achieves 10% performance gain as in Fig. 8. Obviously, with dynamic memory sharing, more data can be accommodated in the integrated architecture for a higher L1D cache hit rate, saving the round-trip cost to L2 or the main memory to benefit both performance and power consumption.

Interference between register and cache data. The integrated architecture works extremely well if there is little interference between the register and cache data. This is important to the replacement policies, especially for the LRU-fair, because both register and cache data are mutually replaceable. Fig. 13 reports the interference in LRU-fair policy, which is given in the form of $x \leftrightarrow y$. For example, $r \leftrightarrow c$ means the data replacement between register and cache. The ratio is calculated by the amount of such interference over the total number of memory write operations. Note that except for $c \leftrightarrow c$, the other types of interference are newly introduced in our integrated architecture.

From the plot, we find that the register is rarely replaced by another register because registers are evenly distributed over all sets in the memory through the re-indexing process. However, the addressing of cache data is much more random so we observe a relatively high $r \leftrightarrow c$ or $c \leftrightarrow c$ rate. More importantly, because the majority of registers are more frequently reused than cache data as observed in Fig. 5(b), the LRU policy is biased to replace the cache data, causing much higher $c \leftrightarrow c$ rate. This makes sense since we want to keep as many live registers on board as possible.

The results imply the harmony between registers and cache data in the integrated memory considering the different on-chip memory sizes and workloads that we have explored. Many of the benchmarks only have a very small aggregated interference rate. For others with larger aggregated interference rate, the primary reason comes from the $c \leftrightarrow c$ interference that is typically one order of magnitude higher than the other types of interference. The harmony mainly relies on the cache alike placement of registers after the re-indexing process. In this way, registers can be flexibly placed within a set to share the space with the cache data. Meanwhile, the LRU policy can well protect the frequently used registers from being replaced by cache data. In summary, our scheme offers a promising solution to virtually enlarge the cache capacity without paying

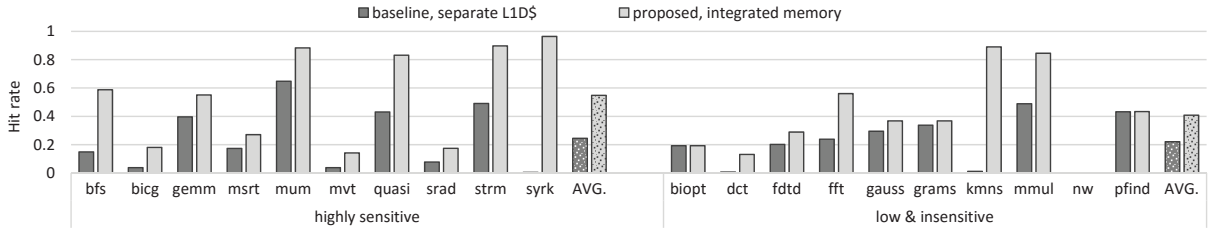

Fig. 12: The cache data hit rates in the baseline and the proposed architecture.

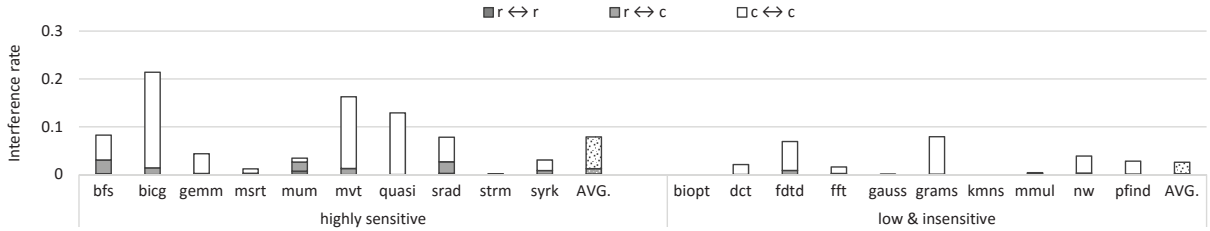

Fig. 13: The interference rate of $r \leftrightarrow r$, $r \leftrightarrow c$, and $c \leftrightarrow c$.

the actual hardware cost.

Bank conflicts. A drawback of the integrated architecture is that it may experience more bank conflicts by accommodating both register and cache data in the same bank. Fig. 14 shows the ratio of the bank conflicts of the proposed architecture normalized to the baseline. From this plot, we see that the conflicts vary over a wide range, but are generally correlated to the memory utilization ratio changes after applying our scheme. For example, "bicg" is found to suffer heavy conflicting because the utilization rate increases quite a lot. In contrast, "mum" shows less bank conflicts because of the moderate utilization improvement using the proposed scheme. In fact, the increased bank conflicts are the nature consequence of an improved bank utilization because now each bank can hold both register and cache data that offers a much increased memory hit rate. Given the fact that the performance is more sensitive to cache misses (typically hundreds of cycles of overhead), the increase in the bank conflicts (typically several cycles of overhead) is actually a worthy tradeoff for a cost-effective high performance design.

On the other hand, the conflict problem can be greatly alleviated by adding more banks with some extra hardware cost. In our evaluation, we still apply the same baseline banking organization for a fair comparison, although may not be optimal for the performance in the integrated architecture.

C. Energy Evaluation

We further evaluate the energy consumption in the integrated memory. The energy reported in the baseline include the separate RF and L1D. The energy reported in the integrated architecture includes the data array, the tag array, register re-indexing and the associate logics. We also consider the L2 energy in both schemes. To simplify the energy calculation, we assume all the missed requests from L1D in the baseline or the integrated memory in the proposed architecture are all captured in L2 without going to the main memory.

Because the proposed integrated architecture has to lookup the tag array to locate both register and cache data, it is possible to increase the power due to: 1) the register re-indexing and tag lookup power, which are not present in the baseline; 2) the cache data accessing power, where every cache access in the integrated architecture has to read a larger data array compared to the smaller separate L1D cache in the baseline.

However, our proposed architecture can save energy owing to the increased hit rate and in turn eliminate many unnecessary L1D lookups upon cache misses that are quite common in the baseline. It also saves energy for not going to the L2 cache. Furthermore, the leakage power is smaller because the footprint of the integrated memory is smaller than the separate L1D and RF.

Fig. 15 reports the energy results, where we see that the energy of the proposed architecture can be significantly reduced by over 40% for the cache-sensitive applications. The energy also improves for the cache-insensitive applications, mainly owing to the reduced execution time that saves the static energy. Although the dynamic power may increase for some applications within the shortened execution time, it is beneficial because the throughput of the chip has been improved with higher energy efficiency.

To be more specific, although the energy of the tag array increases, its impact is negligible because the tag array only takes a very small portion in the total energy. For the cache-sensitive applications, the dynamic energy of the integrated data array can increase notably due to the much increased L1D hit rate, but the tradeoff with the reduced L2 energy eventually leads to the overall energy saving. The total energy is further cut down by the smaller leakage.

Given that the RF and L1D are among the most power-hungry on-chip components in the GPGPUs, we expect the proposed architecture can be very promising for the future efficient and high throughput chip design.

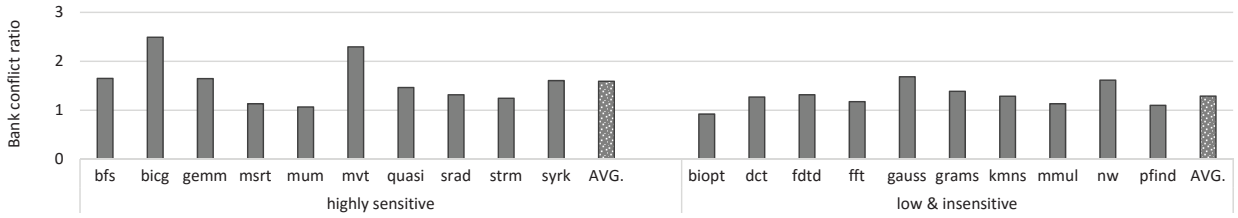

Fig. 14: The normalized bank conflicts.

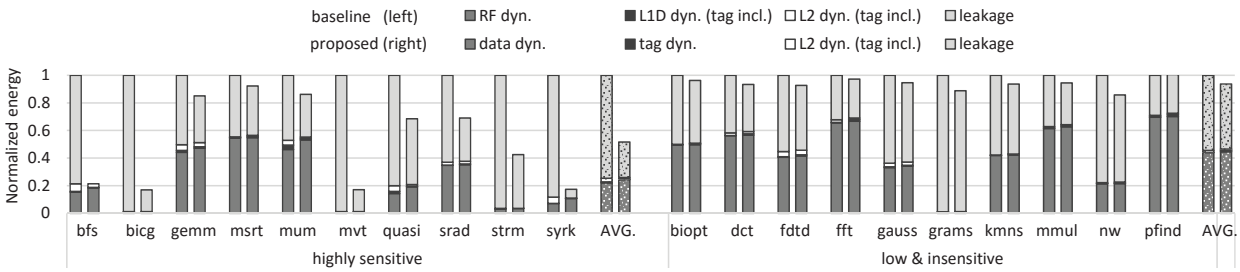

Fig. 15: Energy breakdown and comparison between the baseline and the proposed design.

VI. Related Work

As on-chip memory is critical to GPGPU performance, power and area, there have been intensive studies on it. For example, to improve power or energy efficiency of the on-chip memory, various new memory technologies have been proposed as alternatives to the SRAM-based design. Among them, Yu et. al proposed a customized hybrid SRAM-DRAM RF [26] to save area and power with small performance loss. Goswami et. al proposed a STT-MRAM RF [27] that leverages fast reading to alleviate the long latency in writing. Jing et. al proposed an eDRAM-based on-chip memory with opportunistic refreshing that enables minimum performance loss [11, 28]. Besides, being aware of the large amount of statically wasted registers in RF (i.e. the static holes in this study), power gating techniques were proposed in [15]. Studies [16, 17] pointed out dynamic RF under-utilization at runtime (i.e. the dynamic holes in this study), and [5] found better use of the holes by register compression for power optimization.

Architectural techniques are also intensively studied to improve GPGPU performance via optimizations on separate RF, cache or shared memory. For example, RF caching [4] leveraged an extra small cache in front of the RF for frequent accessed operands. It added nearly 6KB storage per SM and relied on schedulers to enable the benefits from the RF cache structure. Alternative RF structures for control flow optimization were proposed in [29, 30]. There are also other studies on the bank conflict problem in the RF [31, 32] and the shared memory [33].

Recently, L1D cache is drawing growing attention due to the cache thrashing problem that significantly impacts the GPGPU performance. Although originating from CPUs, it is exaggerated in GPGPUs due to the massive concurrent threads. To solve this problem, researchers have proposed various approaches such as the dynamical memory request reordering [6], cache aware thread block scheduling [10], thread block throttling [7], cache bypassing [8] or prioritization [9], statical cache reuse distance analysis [13, 34], and the memory access pattern identification [35] for better cache design. However, these approaches mainly rely on the same cache design wisdom that is already present in CPU, and miss the important opportunity of the interaction between RF and cache as identified in our study.

VII. Conclusion

Facing the insufficient on-chip memory resource in modern GPGPUs, this study proposes a novel integrated on-chip memory architecture. This architecture can smartly identify the resource redundance with the assistance from ISA and compiler, and integrate the RF and L1D cache as a monolithic memory through resource sharing. The architecture can greatly simplify the memory hierarchy design with a uniform addressing and management policy, achieving the cache-emulated RF capability. Experiment results demonstrate significant performance improvement with reduced energy consumption. As the first attempt to reform the on-chip memory hierarchy, we expect this study to prompt new thoughts on the memory designs tailored for high-performance computing in GPGPUs.

Acknowledgment

This work is partly supported by the National Natural Science Foundation of China (Grant No. 61402285, No. 61202026, and No. 61332001), and Program of China National 1,000 Young Talent Plan.

References

[1] NVIDIA Whitepaper, “Nvidia’s next generation CUDA compute architecture: Fermi.”
[2] NVIDIA Whitepaper, “Nvidia’s next generation CUDA compute architecture: Kepler GK110.”
[3] M. Gebhart, S. W. Keckler, B. Khailany, R. Krashinsky, and W. J. Dally, “Unifying primary cache, scratch, and register file

memories in a throughput processor," in *Proceedings of the 45th Annual International Symposium on Microarchitecture*, 2012, pp. 96–106.

[4] 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 *Proceedings of the 38th Annual International Symposium on Computer Architecture*, 2011, pp. 235–246.

[5] S. Lee, K. Kim, G. Koo, H. Jeon, W. W. Ro, and M. Annavaram, "Warped-compression: Enabling power efficient GPUs through register compression," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015, pp. 502–514.

[6] W. Jia, K. Shaw, and M. Martonosi, "MRPB: Memory request prioritization for massively parallel processors," in *Proceedings of the 20th Annual International Symposium on High Performance Computer Architecture*, Feb 2014, pp. 272–283.

[7] X. Chen, L.-W. Chang, C. Rodrigues, J. Lv, Z. Wang, and W. mei Hwu, "Adaptive cache management for energy-efficient GPU computing," in *Proceedings of the 47th Annual International Symposium on Microarchitecture*, Dec 2014, pp. 343–355.

[8] X. Xie, Y. Liang, Y. Wang, G. Sun, and T. Wang, "Coordinated static and dynamic cache bypassing for GPUs," in *Proceedings of the 21st Annual International Symposium on High Performance Computer Architecture*, Feb 2015, pp. 76–88.

[9] S.-Y. Lee, A. Arunkumar, and C.-J. Wu, "CAWA: Coordinated warp scheduling and cache prioritization for critical warp acceleration of GPGPU workloads," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015, pp. 515–527.

[10] T. Rogers, M. O'Connor, and T. Aamodt, "Cache-conscious wavefront scheduling," in *Proceedings of the 45th Annual International Symposium on Microarchitecture*, Dec 2012, pp. 72–83.

[11] N. Jing, Y. Shen, Y. Lu, S. Ganapathy, Z. Mao, M. Guo, R. Canal, and X. Liang, "An energy-efficient and scalable eDRAM-based register file architecture for GPGPU," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, June 2013, pp. 344–355.

[12] J. Lim, N. B. Lakshminarayana, H. Kim, W. Song, S. Yalamanchili, and W. Sung, "Power modeling for GPU architectures using McPAT," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 19, no. 3, pp. 26:1–26:24, Jun. 2014.

[13] C. Nugteren, G.-J. van den Braak, H. Corporaal, and H. Bal, "A detailed GPU cache model based on reuse distance theory," in *Proceedings of the 20th International Symposium on High Performance Computer Architecture*, Feb 2014, pp. 37–48.

[14] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying GPU microarchitecture through microbenchmarking," in *Proceedings of the International Symposium on Performance Analysis of Systems Software (ISPASS)*, March 2010, pp. 235–246.

[15] M. Abdel-Majeed and M. Annavaram, "Warped register file: A power efficient register file for GPGPUs," in *Proceedings of the 19th Annual International Symposium on High Performance Computer Architecture*, 2013, pp. 412–423.

[16] H. Jeon and M. Annavaram, "GPGPU register file management by hardware cooperated register reallocation," in *Computer Engineering Technical Report Number CENG-2014-05 in University of Southern California*, 2014.

[17] H. Jeon, G. S. Ravi, N. S. Kim, and M. Annavaram, "GPU register file virtualization," in *Proceedings of the 48th Annual International Symposium on Microarchitecture*, 2015, pp. 420–432.

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

[19] A. W. Appel, *Modern Compiler Implementation in C: Basic Techniques*. Cambridge University Press, 1997.

[20] NVIDIA Whitepaper, "Parallel thread execution ISA version 3.0," 2012.

[21] NVIDIA, "CUDA SDK," https://developer.nvidia.com/cuda-toolkit.

[22] R. S. S. A. Scott Grauer-Gray, Lifan Xu and J. Cavazos, "Auto-tuning a high-level language targeted to GPU codes," in *Proceedings of Innovative Parallel Computing (InPar)*, 2012.

[23] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, 2009, pp. 44–54.

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

[25] "CACTI: An integrated cache and memory access time, cycle time, area, leakage, and dynamic power model," http://www.hpl.hp.com/research/cacti/.

[26] W.-k. S. Yu, R. Huang, S. Q. Xu, S.-E. Wang, E. Kan, and G. E. Suh, "SRAM-DRAM hybrid memory with applications to efficient register files in fine-grained multi-threading," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, 2011, pp. 247–258.

[27] B. C. Nilanjan Goswami and T. Li, "Power-performance co-optimization of throughput core architecture using resistive memory," in *Proceedings of the Annual IEEE International Symposium on High Performance Computer Architecture*, 2013.

[28] N. Jing, L. Jiang, T. Zhang, C. Li, F. Fan, and X. Liang, "Energy-efficient eDRAM-based on-chip storage architecture for GPGPUs," *IEEE Trans. Computers*, vol. 65, no. 1, pp. 122–135, 2016.

[29] W. Fung, I. Sham, G. Yuan, and T. Aamodt, "Dynamic warp formation and scheduling for efficient GPU control flow," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec 2007, pp. 407–420.

[30] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt, "Improving GPU performance via large warps and two-level warp scheduling," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011, pp. 308–317.

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

[32] N. Jing, S. Jiang, S. Chen, J. Zhang, L. Jiang, C. Li, and X. Liang, "Bank stealing for a compact and efficient register file architecture in GPGPU," *IEEE Trans. VLSI Systems*, 2016.

[33] C. Gou and G. Gaydadjiev, "Elastic pipeline: addressing GPU on-chip shared memory bank conflicts," in *Proceedings of the 8th Conference on Computing Frontiers*, 2011, p. 3.

[34] S. Das, T. M. Aamodt, and W. J. Dally, "Reuse distance-based probabilistic cache replacement," *ACM Trans. Archit. Code Optim.*, vol. 12, no. 4, pp. 33:1–33:22, Oct. 2015.

[35] W. Jia, K. A. Shaw, and M. Martonosi, "Characterizing and improving the use of demand-fetched caches in GPUs," in *Proceedings of the 26th ACM International Conference on Supercomputing*, 2012, pp. 15–24.