# LTRF: Enabling High-Capacity Register Files for GPUs via Hardware/Software Cooperative Register Prefetching

Mohammad Sadrosadati[1,2] Amirhossein Mirhosseini[3] Seyed Borna Ehsani[1] Hamid Sarbazi-Azad[1,4]
Mario Drumond[5] Babak Falsafi[5] Rachata Ausavarungnirun[6] Onur Mutlu[2,6]

[1]Sharif University of Technology [2]ETH Zürich [3]University of Michigan
[4]IPM [5]EPFL [6]Carnegie Mellon University

## Abstract

Graphics Processing Units (GPUs) employ large register files to accommodate all active threads and accelerate context switching. Unfortunately, register files are a scalability bottleneck for future GPUs due to long access latency, high power consumption, and large silicon area provisioning. Prior work proposes hierarchical register file, to reduce the register file power consumption by caching registers in a smaller register file cache. Unfortunately, this approach does not improve register access latency due to the low hit rate in the register file cache.

In this paper, we propose the Latency-Tolerant Register File (LTRF) architecture to achieve low latency in a two-level hierarchical structure while keeping power consumption low. We observe that compile-time interval analysis enables us to divide GPU program execution into intervals with an accurate estimate of a warp's aggregate register working-set within each interval. The key idea of LTRF is to prefetch the estimated register working-set from the main register file to the register file cache under software control, at the beginning of each interval, and overlap the prefetch latency with the execution of other warps. Our experimental results show that LTRF enables high-capacity yet long-latency main GPU register files, paving the way for various optimizations. As an example optimization, we implement the main register file with emerging high-density high-latency memory technologies, enabling 8× larger capacity and improving overall GPU performance by 31% while reducing register file power consumption by 46%.

***Keywords*** GPUs, Register File Design, Latency Tolerance, Energy Efficiency, Memory Technology, Memory Latency

**ACM Reference format:**
Sadrosadati et al. LTRF: Enabling High-Capacity Register Files for GPUs via Hardware/Software Cooperative Register Prefetching. In *Proceedings of The 23rd ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Williamsburg, VA, USA., March 24–28, 2018 (ASPLOS '18),* 14 pages.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
*ASPLOS '18, March 24–28, 2018, Williamsburg, VA, USA.*
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-4911-6/18/03…$15.00
https://doi.org/10.1145/3173162.3173211

## 1 Introduction

Graphics Processing Units (GPUs) are commonly-used accelerators, optimizing silicon organization with dense arithmetic for data-parallel workloads. Modern GPU microarchitecture relies on managing execution resources for a large number of Single-Instruction-Multiple-Data (SIMD) threads to exploit this arithmetic density and overlap the long memory access latency with computation [55]. Unfortunately, the maximum parallelism in GPUs is fundamentally limited by the register file capacity as the register file must accommodate all simultaneously running threads [3, 19, 20, 21, 39, 48, 80].

GPU register files face the difficult challenge of optimizing latency, bandwidth, and power consumption, while having maximal capacity. Prior work proposes increasing the register file capacity in various ways: compression [39], virtualization [25, 78], or silicon technologies for high-density memory cells [27, 28, 43, 45, 46, 48, 80]. While such proposals increase capacity without sacrificing power consumption, they typically result in higher register access latencies.

Register file caching [19, 20] is a promising approach to enhancing capacity while lowering power consumption and effective access latency. Unfortunately, existing proposals for register file caching do *not* achieve high enough hit rates in the register cache due to *three* key problems. First, the high degree of thread-level parallelism (TLP) in GPUs causes threads to displace each other's registers in the cache. Second, registers house temporary values that are often renamed, which reduces temporal locality in the cache. Third, because register names are *not* spatially correlated, there is no spatial locality in a register cache. Due to these reasons, register file caching is ineffective at hiding latency in GPUs (§ 6).

Our goal is to improve the effectiveness of register file caching in GPUs. To this end, we observe that registers can be effectively prefetched into the register cache using compile-time interval analysis to hide the long access latency of the main register file. An Interval is a subgraph in a program's control-flow graph that has a *single* entry point. Intervals have been widely used by optimizing compilers to identify loops [22]. We use interval analysis and software prefetching to fetch the entire set of required registers of an interval into the register cache and thus avoid the main register file access latency during the execution of the interval.

We propose the *Latency-Tolerant Register File (LTRF)*, a two-level hierarchical register file that employs a low-latency/low-power first-level register-file cache backed up by a high-latency/high-capacity second-level main register file. LTRF uses a compiler-driven software mechanism to prefetch a warp's register working-set into the register cache at the start of an interval. By fetching *all* registers in the working-set together and overlapping the prefetch latency of one warp with the execution of another, LTRF hides a substantial

fraction of the access latency of the main register file during the execution of the interval.

By using LTRF, we enable high-capacity yet long-latency main register files, paving the way for various optimizations. As an example optimization, we implement the main register file with high-density emerging memory technologies, e.g., domain wall memory [4, 5, 48, 59, 69, 76], enabling 8× larger capacity and improving overall GPU performance by 31% while reducing register file power consumption by 46%. In contrast, the state-of-the-art register file caching schemes reduce GPU performance by 14%, on average, if the register file is enlarged by 8×, as prior designs do *not* focus on tolerating the latency of the main register file.

This paper makes the following contributions

- We show that prior proposals for register file caching do *not* achieve high enough hit rates to effectively hide the long latencies of large main register files (§ 6).
- We introduce LTRF, a latency-tolerant hierarchical register file design, which enables high-capacity yet long-latency main register files. The key idea is to 1) estimate the register working set of a program's execution during an interval, using compile-time interval analysis, 2) prefetch the estimated register working-set from the main register file to the register-file cache under software control, at the beginning of each interval, and overlap the prefetch latency with the execution of other warps.
- Our evaluations show that an optimized version of LTRF, when implemented with an 8× larger yet 6.3× slower main register file, improves overall GPU throughput by 31%, on average (up to 86%). LTRF performance is within 5% of an ideal 8×-capacity main register file that has no latency overhead.

## 2 Background and Motivation

Figure 1 illustrates a conventional GPU register file architecture [44] in a streaming multiprocessor (SM). To accommodate a large number of active threads, a GPU employs a register file of megabytes in size. For example, GP100 (NVIDIA Pascal) has has a register file of 14.3 megabytes in total [57]. The register file is heavily banked (16 banks) and it allows concurrent accesses from many threads (up to 512 threads). Each bank stores registers from multiple warps. When the GPU issues an instruction, an operand collector concurrently accesses and gathers data associated with each thread in the issued warp's instruction through an arbiter and a large and wide crossbar, as shown in Figure 1. The warp scheduler arbitrates among *ready* warps (i.e., a warp whose operands are collected) and issues the warp's instruction to the SIMD units.

In this section, we demonstrate the increasing demand for larger register file capacity, analyze shortcomings of prior register caching mechanisms for GPUs, and motivate the case for a design that provides high capacity without significantly increasing power consumption, on-chip die area, or access latency exposed to the GPU core.

### 2.1 Factors that Limit GPU Performance

When a warp encounters a long-latency memory instruction, the GPU selects another *ready* warp to be scheduled for execution, in order to prevent the GPU core from stalling. While the applications with high TLP are more likely to contain more ready warps and are able to hide long-latency stalls more effectively, these applications

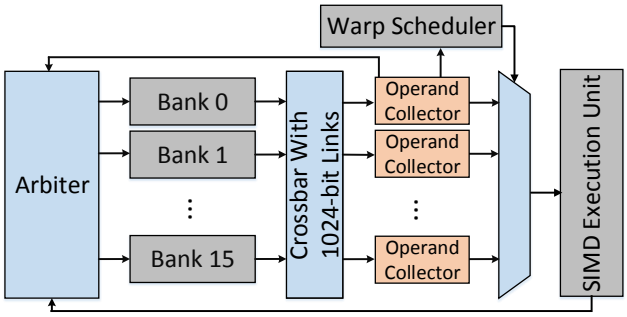


**Figure 1.** Conventional GPU register file architecture.

with high TLP demand a large register file in order to realize their maximum TLP. To illustrate the impact of the register file size on an application's TLP, we recompile 35 workloads in CUDA SDK [10], Rodinia [14], and Parboil [72] benchmark suites with the *maxregcount* attribute (i.e., the attribute that enables the use of the maximum number of registers for each GPU function, i.e., 64 and 256 for Fermi and Maxwell respectively) enabled in the NVIDIA GPU compiler, *nvcc*. Doing so enables us to measure the number of registers applications would require if there were *no* register file size constraints.

Table 1 reports the average and maximum register file capacity needed for our benchmarks to achieve the maximum TLP provided by the two GPU products. This experiment shows that a larger register file would directly translate into a larger number of executing threads, thereby increasing TLP, on average. The table corroborates our intuition that TLP is indeed limited by the number of registers and many applications benefit from compiler optimization when given a larger register file [47, 52, 84]. The results also show that the recent version of the CUDA compiler used for Maxwell employs more aggressive compiler optimization techniques (e.g., loop unrolling) and as such enhances register usage and TLP compared to Fermi.

| GPU (baseline register file size) | Average required register file size | Maximum required register file size |
|---|---|---|
| Fermi (128KB) | 184KB (1.4×) | 324KB (2.5×) |
| Maxwell (256KB) | 588KB (2.3×) | 1504KB (5.9×) |

**Table 1.** The average and maximum register file capacity required to maximize TLP for 35 workloads in CUDA SDK [10], Rodinia [14], and Parboil [72] benchmark suites in the NVIDIA Fermi and Maxwell architectures.

### 2.2 Register File Scalability

While modern GPUs integrate more execution resources with increases in silicon density and memory bandwidth in each chip generation, the register file accounts for an increasingly larger fraction of on-chip storage, as shown in Figure 2. For NVIDIA Pascal [57], more than 60% of the on-chip storage area, amounting to 14.3 MB is dedicated to the register file. GPU register files face the difficult challenge of optimizing latency, bandwidth, and power consumption, while having maximal capacity [2, 19, 20, 23, 25, 27, 28, 39, 43, 45, 46, 48, 65, 66, 78, 79, 80]. Larger register files are slower, take up more silicon area and consume more power. Increasing concurrency by adding more banks exacerbates complexity and power consumption with the addition of a larger crossbar. Prior

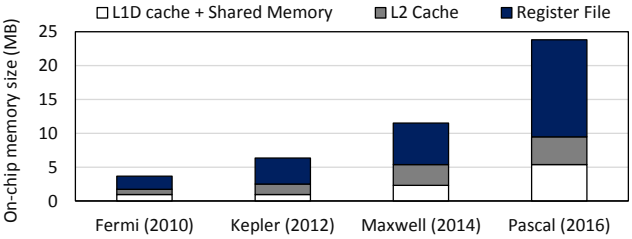


**Figure 2.** Capacity of on-chip memory components across generations of NVIDIA GPUs from 2010–2016.

work attempts to reduce the power consumption of the register file while keeping the register access latency almost unchanged. As a result, the reduction in the power consumption is limited by the access latency of the register file. In this section, we measure the impact of various register file design parameters and configurations on register file access latency and overall GPU throughput.

Table 2 illustrates register file designs with varying parameters, including cell technology, number of banks, bank size, and network topology, relative to a baseline high performance SRAM-based design shown in Configuration #1. The table also presents results for emerging memory cell technologies that enable a larger trade-off space between area, power and latency. We use high-performance (HP) CMOS, low-standby-power (LSTP) CMOS, tunnel-field-effect transistors (TFET), and domain-wall memory (DWM) for the cell technology [4, 5, 12, 17, 18, 41, 43, 48, 50, 51, 59, 60, 69, 71, 74, 76, 77, 81]. To obtain these results, we first use CACTI [51] (the non-pipelined register file bank models) and NVSim [17] to extract timing, area and power, and then feed them as parameters to GPGPU-Sim [10] to measure the average register file access latencies. The results include queuing delays incurred due to bank conflicts (Our system configuration is presented in § 5). Note that we use the flattened butterfly topology [35] to reduce the overhead of the crossbar network when we increase the number of banks by 8× in our implementations. We make two key observations from Table 2. First, register file designs (such as design #7) that minimize area and power consumption while optimizing for capacity (i.e., bits/area) exhibit higher access latency. Second, while some alternative cell technologies (e.g., DWM [48, 76]) can dramatically improve capacity and power consumption, they incur prohibitively long access latencies (e.g., as long as 6.3× compared to the baseline register file).

To illustrate the potential benefit of using a large register file, Figure 3 plots performance (in IPC) for a high-capacity register file implemented using TFET-SRAM and an "Ideal TFET-SRAM", which has the same capacity as TFET-SRAM, but also the same latency as the baseline register file, normalized to the baseline register file from Table2.[1] We categorize our workloads into two groups: *register-insensitive* and *register-sensitive*. Register-insensitive workloads are the ones where the register file size is *not* the bottleneck for higher TLP; i.e., increasing the register file size does *not* improve TLP. We make two key observations. First, we find that the Ideal TFET-SRAM, which increases the register file size from 256KB to 2MB *without* increasing the register file access latency, improves IPC throughput by 10%-95% (37%, on average) for register-sensitive workloads. We find that the IPC improvements are due to both more registers per thread and more warps executing in parallel. Second, when real latencies are modeled, much of the gain from higher capacity and TLP is offset by higher latency, and overall performance *reduces* despite the higher register file capacity. We conclude that register file access latency is important for performance and should be kept in check while increasing register file capacity.

[1] We choose a 2MB TFET-SRAM register file as it consumes a similar amount of power as our baseline 256KB register file (see Table 2).

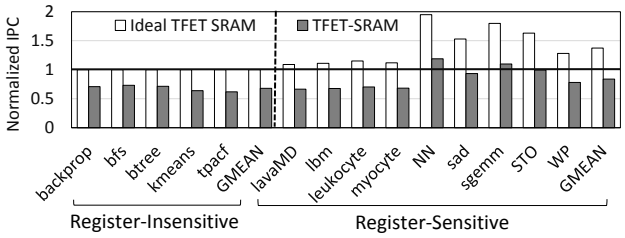


**Figure 3.** Performance effect of increasing the register file size by 8× using ideal TFET-SRAM and real TFET-SRAM designs, normalized to the IPC of the baseline architecture with a 256KB register file.

### 2.3 Register File Caching

One method to increase the size of the register file while keeping access latency low is to cache registers in a smaller structure, i.e., register file caching. Although there is significant previous work on register file caches for CPUs [11, 15, 16, 31, 54, 58, 70, 73, 86, 87], and vector processors [13, 32, 63], register file caching has *not* been thoroughly investigated in GPU designs. Gebhart et al. [19] are the first to introduce register file caches for GPUs to filter some of the accesses to the main register file and thus reduce the dynamic access energy of the main register file. The authors' design works almost the same way as a conventional cache structure and exploits temporal locality. However, as Figure 4 shows, for a 16KB register cache, the register cache hit rate is low: between 8% and 30%.

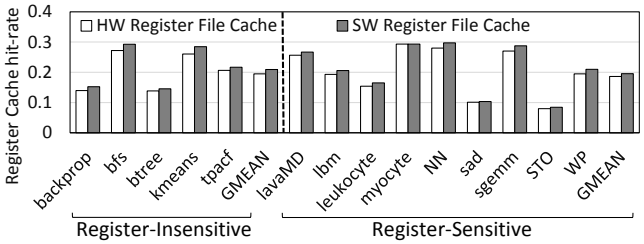


**Figure 4.** Hit rate in hardware [19] and software [20] register file caches.

We find that the hardware register cache hit rate is low due to the following reasons:

1. Different warps can displace each other's registers in the cache due to the high warp switching rate in GPUs. This thrashing effect is also observed in SMs' local data caches [6, 7, 8, 9, 19, 24, 29, 30, 33, 34, 38, 42, 49, 62, 64, 67, 68, 75, 82, 83].
2. We find that many registers are used to only communicate results between a few instructions. As a result, these registers do *not* have good temporal locality.
3. There is no notion of "spatial locality" in register accesses (i.e., there is no logical order among different registers).

| Config. | Cell Technology | #Banks | Bank Size | Network | Cap. | Area | Power | Cap./Area | Cap./Power | Latency |
|---|---|---|---|---|---|---|---|---|---|---|
| #1 | HP SRAM | 1× | 1× | Crossbar | 1× | 1× | 1× | 1× | 1× | 1× |
| #2 | HP SRAM | 1× | 8× | Crossbar | 8× | 8× | 8× | 1× | 1× | 1.25× |
| #3 | HP SRAM | 8× | 1× | F. Butterfly | 8× | 8× | 8× | 1× | 1× | 1.5× |
| #4 | LSTP SRAM | 1× | 8× | Crossbar | 8× | 8× | 3.2× | 1× | 2.5× | 1.6× |
| #5 | LSTP SRAM | 8× | 1× | F. Butterfly | 8× | 8× | 3.2× | 1× | 2.5× | 2.8× |
| #6 | TFET SRAM | 8× | 1× | F. Butterfly | 8× | 8× | 1.05× | 1× | 7.6× | 5.3× |
| #7 | DWM | 8× | 1× | F. Butterfly | 8× | 0.25× | 0.65× | 32× | 12× | 6.3× |

**Table 2.** Various register file designs with different configurations; all the numbers including number of banks (1× = 16), bank size (1× = $16KB$), capacity, area, power consumption, capacity per area, capacity per power, and access latency are normalized to the baseline GPU register file with 256KB size and 16 banks.

Follow-up work [20] proposes a software-managed hierarchical register file (SHRF) that aims to reduce data movement between the main register file and the register cache. However, as the main objective is to reduce dynamic energy consumption of the baseline large monolithic register file, the authors [20] aim to reduce the total number of accesses to the main register file, regardless of whether or not those accesses occur during the execution of a warp. In particular, SHRF reduces the extra register file accesses caused by register file cache write-back/reloads, by adding specialized instructions, aided by a new register allocation mechanism, to manage register movement. However, Figure 4 shows that the software approach does *not* significantly improve the hit rate compared to a baseline hardware register cache [19] as it mostly focuses on reducing the number of background (i.e., write-back/reload) register accesses, rather than accesses that are needed by program instructions.

### 2.4 Summary and Goals

In this work, we leverage two observations we have provided in this section. First, the register file is one of the limiting factors in the scalability of GPUs in terms of TLP. Second, making the register file considerably larger is very difficult without sacrificing either latency or power consumption. Register caching can reduce the register access latency and thus enable aggressive power optimization techniques without degrading GPU performance. However, register caching has *not* been thoroughly studied in the context of GPUs and the existing schemes mainly aim to reduce power consumption, rather than completely hide the main register file access latency. Therefore, these designs are inefficient as they do not offer high register cache hit rates. In fact, they hurt performance if used with considerably slow main register files (see § 6).

In this paper, we aim to architect a *latency-tolerant* hierarchical register file for GPUs that can have very high capacity. Our goal is to 1) enable very high-capacity yet also high-latency main register files, while improving performance, and thus 2) open the design space for many power/area optimization techniques in the main register file that likely increase the register access latency (and thus would otherwise be unacceptable).

## 3 LTRF

To make the register file very high capacity and at the same time latency-tolerant, we propose a new register file caching mechanism that aims to 1) bring the warps' registers into the register file cache *before* they are accessed by the warps (i.e., register prefetching) and 2) service all register accesses from the register file cache. As a result, the warps see the latency of a fast register cache and *not* the slow main register file. We find that a near-perfect register prefetching mechanism can be implemented based on two key observations. First, the register working-set is known at compile-time as there is no indirection or aliasing in register accesses. Second, long register access latency can be hidden by the execution of other active warps.

LTRF takes advantage of these two observations that enable a new register prefetching scheme. § 3.1 provides an overview of our register prefetching scheme. § 3.2 and § 3.3 provide an overview of the architectural and compiler support required for our software-driven prefetching scheme, respectively.

### 3.1 Register Prefetching Scheme

We define a PREFETCH operation to specify which registers should be prefetched from the main register file. A PREFETCH operation brings the register working-set of a subgraph of the application control flow graph (CFG) into the register cache. The working-set is composed of the registers that, depending on the dynamic control flow, *might be* accessed between two PREFETCH operations. We call subgraphs of the CFG created by PREFETCH operations (bounded by PREFETCH operations) *prefetch subgraphs*. Finding an optimal placement of PREFETCH operations is not only impossible in polynomial time, but also requires information available only during runtime because of dynamic warp interleavings. We propose a heuristic algorithm that employs the concept of *intervals* [22], subgraphs of the CFG with a single entry point, which offers compile-time analysis within a reasonable amount of time. We modify the classic interval analysis algorithm, used to find the subgraphs of the CFG with a single entry point, and introduce the concept of *register-intervals* as suitable prefetch subgraphs for prefetching registers. A *register-interval* is a subgraph of the CFG that 1) has a single control flow entry point and 2) requires, at most, a given number of registers.

Our scheme brings the register working-set into the cache at the beginning of each register-interval and *guarantees* that *all* register accesses made inside that register-interval will be serviced from the register file cache.

### 3.2 Architectural Support

To reduce the register file cache size, we limit the number of active warps that run concurrently and maintain a pool of inactive warps; the inactive warps remain dormant and are not allocated space in the register file cache. Furthermore, we partition our register file cache and allocate each partition to an active warp, thus preventing active warps from contending for register file cache space, and

thus from evicting each other's registers. We size the dedicated caching space for each warp according to the maximum number of registers the warp can access throughout the execution of a prefetch subgraph. This parameter also sets an upper bound for the size of a prefetch subgraph working-set. By ensuring no register cache evictions occur during the execution of a prefetch subgraph, we guarantee that register movement happens only with PREFETCH operations or when a warp becomes active/inactive.

We deploy a two-level warp scheduler, similar to the one used in [19, 53], to schedule execution of active warps. The scheduler issues instructions from active warps in a fair manner (e.g., round-robin). Whenever a warp encounters a long latency operation, such as a data cache miss, it becomes inactive and gets replaced by another one from the active pool. The two-level scheduler enables the use of a smaller register file cache that needs to accommodate only the working-sets of the *active* warps, and a warp's register working-set is swapped in and out of the register file cache as warp becomes active/inactive.

Reducing the number of *active* warps provides two positive benefits: it 1) does *not* limit TLP since inactive warps still maintain live state in the *main* register file, and thus can be quickly activated, 2) can potentially improve performance by reducing the L1 data cache thrashing effect and by preventing *all* warps from stalling at the same time [33, 34, 53, 62, 68]. In LTRF, warp activations are not cost-free as the register working-set of the inactive warp needs to be prefetched before the warp becomes active. Hence, if we cannot hide the warp activation latency, we might negatively affect performance. In § 6.3, we quantitatively show that this is not the case. LTRF requires a small number of active warps to hide the warp activation latency, allowing a GPU to tolerate higher latency accesses to the main register file (We discuss the design of the main register file and the register cache in detail in § 4.1).

PREFETCH operations use bit-vectors to identify the registers that should be cached for each prefetch subgraph, enabling support for various cache sizes. The PREFETCH bit-vector size is equal to the maximum number of registers the CUDA compiler can allocate to a thread. For example, in the latest CUDA versions, the compiler can allocate up to 256 registers to each thread, requiring a 256-bit vector for each PREFETCH operation. The instruction fetch unit needs to know in advance when it is going to process a PREFETCH bit-vector. We consider two approaches. The first embeds an extra bit in each instruction to indicate whether a PREFETCH bit-vector follows that instruction. Prior work [20] has similar requirements and the authors show that, in general, the cost of embedding the extra bit is negligible. The second approach is to add an explicit instruction that is always followed by the bit-vector. We show in §4.3 that code-size and performance overheads are negligible with either of the approaches.

When a warp becomes inactive, we must keep track of which registers should be written back and refetched once the warp becomes active again. In LTRF, we simply write back and refetch the *entire* register working-set of the active prefetch subgraph.

In order to improve the efficiency of the basic LTRF design, we devise operand-liveness aware LTRF (called LTRF+), which considers the liveness of the registers to save register file cache space. The key idea of LTRF+ is to avoid writing-back/re-fetching dead registers. To this end, each read operand has to be extended with an additional bit, called the *dead operand bit* as defined in [19], which indicates whether the corresponding operand will be dead after the execution of the corresponding instruction. This information can be conservatively known at compile-time, using static liveness analysis. These bits are used to update the liveness bit vector. The liveness bit vector keeps track of the liveness status of all registers at the current point of execution. A register becomes live when it is written to and dead when an instruction indicates it is dead via the dead operand bit. When a warp becomes inactive, LTRF+ writes back only the live registers to the main register file. When a warp becomes active, LTRF+ fetches only the live registers from the main register file. LTRF+ does *not* read the dead registers from the main register file since their first access, if any, will be a write, and LTRF+ needs to only allocate space for them in the register file cache.

### 3.3 Compiler Support

When a warp reaches the beginning of a *prefetch subgraph*, it is paused until all of its working-set registers are loaded into the register cache. Therefore, PREFETCH operations may have long latencies that can potentially impose large performance overheads, and hence, they should happen infrequently. In order to address this issue, we introduce *register-intervals* as effective prefetch subgraphs and partition the CFG into register-intervals. A register-interval is a subgraph of the CFG with only two constraints. First, it needs to have only one control flow entry point. Second, the number of registers used in a register-interval should *not* exceed the size of a partition in the register cache.[2] The primary difference between register-intervals and other similar concepts, such as *strands* [20], is that complex control flow structures (e.g., backward branches) are allowed inside a register-interval and they do not cause the termination of the register-interval. By relaxing such constraints, register-intervals provide two main benefits. First, register-intervals can have more static instructions and thus the number of PREFETCH operations can be minimized. Second, our mechanism aims to fit a loop within a single register-interval in order to increase the dynamic length of the register-intervals.

We employ classic interval analysis methods [22] to form register-intervals. The original interval concept [22], used in classic compiler algorithms, partitions the CFG into smaller disjoint subgraphs, each with exactly one entry point. These intervals are typically used to identify loops and determine if the CFG is reducible. We constrain the formation algorithm to guarantee that the register working-set of each interval can fit into a register file cache partition. As a result, the register-intervals constructed by our algorithm might be smaller than the intervals formed by the original algorithm and may terminate at arbitrary points. Thus, we modified the original algorithm to construct intervals at arbitrary starting points.

Our register-interval formation algorithm is a multi-pass algorithm. Algorithm 1 shows the first pass. The algorithm tries to compose register-intervals with as many basic blocks as possible. Therefore, it initializes the first register-interval with the entry basic block (line 8) and iteratively attempts to add subsequent blocks to it (lines 9-25). A candidate block must satisfy two conditions to be successfully added: 1) it must be entered only from the current register-interval, 2) the register file cache space allocated for a warp must be enough to house both the active registers already in the

[2] We provide dedicated space for each active warp in the register file cache.

**Algorithm 1** Register-Interval Formation: Pass 1.

**Input: Application Control Flow Graph (CFG)**
**Output: Register-Interval CFG**

```
1:  Initialize:
2:  for each basic block : BB do
3:     BB.input_list ← empty() // List of all register in the register cache at
       the begining of BB
4:     BB.reg_interval ← Unknown
5:  end for
6:  Working-Set ← empty()
7:  entry_block.reg_interval ← new reg_interval() // Each CFG has an
       entry basic block
8:  Working-Set.insert(entry_block)

9:  while (!Working-Set.empty()) do
10:    BB ← a basic block from Working-Set
11:    TRAVERSE(BB)
12:    i ← BB.reg_interval
13:    while (∃ basic block h for which h.reg_interval==Unknown & all of h
       predecessors belong to i & union(output_list of all S
       predecessors).size()≤N) // N is the maximum number of registers al-
       lowed in the register-interval (i.e., size of a partition in the register file
       cache)    do
14:        h.reg_interval ← i
15:        h.input_list ← union(output_list of all h predecessors)
16:        TRAVERSE(h)
17:    end while
18:    for each S ∈ i.successors() do
19:       if (S.reg_interval==Unknown) then
20:          S.reg_interval ← new reg_interval()
21:          S.input_list ← empty()
22:          Working-Set.insert(S)
23:       end if
24:    end for
25: end while

26: procedure TRAVERSE(BB)
27:    register_list ← BB.input_list
28:    for each instruction in BB do
29:       update register_list
30:       if (register_list.size()>N) then
31:          cut BB and introduce a new basic block : BB1
32:          BB1.reg_interval ← new reg_interval()
33:          BB1.input_list ← empty()
34:          Working-Set.insert(BB1)
35:          BB.output_list ← register_list // List of all registers in the regis-
       ter file cache at the end of BB
36:          exit
37:       end if
38:    end for
39: end procedure
```

register-interval and the ones added by the new block. The algorithm stops when it cannot find any basic blocks that meet these conditions (line 13). After it finishes the first register-interval, it creates new register-intervals out of all the basic blocks with incoming edges from that register-interval (lines 18-24). When a register-interval is completely formed, all of the basic blocks that have incoming edges from that register-interval become new register-intervals' headers. If a single basic block's active registers do *not* fit into the remaining register file cache space for that register-interval, the basic block is split across two or more register-intervals (lines 30-37). We also split the basic blocks at function calls (each function call becomes a separate register-interval). The first pass ends when all basic blocks are assigned to register-intervals. After the first pass, the CFG is transformed into a Register-Interval CFG where the nodes represent the register-intervals rather than basic blocks.

Algorithm 2 shows the second pass of our register-interval formation algorithm. This pass reduces the Register-Interval CFG into

**Algorithm 2** Register-Interval Formation: Pass 2.

**Input: Register-Interval CFG**
**Output: Reduced Register-Interval CFG**

```
1:  Initialize:
2:  for each register-interval : i do
3:     i.register-interval ← Unknown
4:  end for
5:  Working-Set ← empty()
6:  entry_register-interval.next_level_register-interval ← new
       next_level_register-interval()
7:  Working-Set.insert(entry_register-interval)

8:  while (!Working-Set.empty()) do
9:     i ← a register-interval from Working-Set
10:    ii ← i.next_level_register-interval
11:    ii.register_list ← i.register_list
12:    while (∃ register-interval h for which h.next_level_register-
       interval==Unknown & all of h predecessors belong to ii & union(reg_list
       of all h predecessors).size()≤N) // N is the maximum number of registers
       allowed in the register-interval (i.e., size of a partition in the register
       file cache)    do
13:        h.next_level_register-interval ← ii
14:        ii.register_list ← union(ii.register_list & h.register_list)
15:    end while
16:    for each S ∈ ii.successors() do
17:       if (S.next_level_register-interval==Unknown) then
18:          S.next_level_register-interval ← new next_level_register-
       interval()
19:          Working-Set.insert(S)
20:       end if
21:    end for
22: end while
```

a smaller number of register-intervals. It works similarly to the first pass, with the difference that it never splits register-intervals. Instead, it merges two register-intervals if 1) one of them can be reached only from the other and 2) the union of their register working-sets still fits into the allocated register file cache space (lines 12-15). The second pass is repeated until the CFG can *not* be reduced anymore.

The only control flow constraint imposed by intervals is that a node can only join an interval when all of the incoming edges to the node come from that interval. As a result, backward edges and thus loop headers always create new intervals.[3] This key feature of intervals makes them ideal subgraphs for our purpose. By starting a new register-interval for each loop, Algorithm 2 maximizes the probability that an entire loop can fit into the register-interval, thereby minimizing the number of PREFETCH operations to one for the entire loop.

The primary role of the second pass is to prevent the mentioned control flow constraint from splitting large register-intervals into multiple smaller ones. As an example, consider the two nested loops in Figure 6. Assuming the entire register working-set of the graph fits in the register file cache, in the first pass, basic block "A" forms register-interval 1. Basic block "B" cannot be merged with register-interval 1 as it has another incoming edge from basic block "C". Therefore, basic block "B" forms a new register-interval, named register-interval 2. Basic block "C" can be merged into register-interval 2 as basic block "C" has only one incoming edge from

[3]This is true only for reducible CFGs with natural loops where the loop has only one entry point [22]. However, this is usually the case as standard languages can usually only represent natural loops (except in some cases with irregular control flow structures, such as GOTO) and compiler infrastructures only produce reducible CFGs [37].

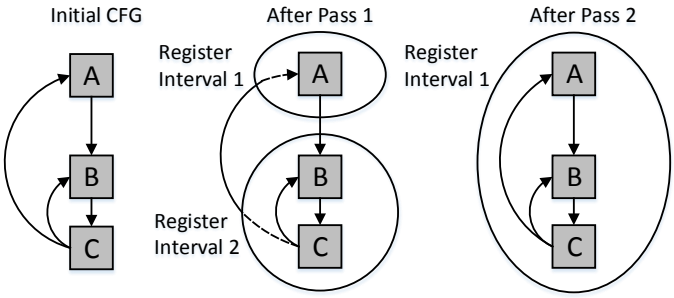


**Figure 6.** Register-interval formation for a simple nested loop example. A,B,C represent basic blocks.

register-interval 2. As a result, the innermost loop becomes a separate register-interval but it cannot be merged into the outermost loop. In the second pass, register-interval 1 can be merged into register-interval 2 as register-interval 1 has only one incoming edge from register-interval 2. Thus, the whole outermost loop can be reduced to a single register-interval. Each repetition of the second pass of the algorithm reduces the depth of a nested loop by one if the resulting register working-set is small enough to fit in the register file cache.

We open source the C implementation of Algorithms 1 and 2 in [1].

## 4 Hardware Implementation

In this section, we discuss the hardware implementation of LTRF in detail.

### 4.1 Register File Microarchitecture

**Register File Cache.** Figure 5 illustrates an example LTRF architecture. We show our added components to the baseline register file architecture in orange color. The register file cache is composed of *#Registers_per_Interval* banks (e.g., 16 banks in the figure) where each bank hosts *#Active_Warps* registers (e.g., 8 1024-bit registers in the figure). LTRF interleaves registers belonging to a single warp across the cache banks, and hence, each register bank houses no more than one register of a warp. Register file cache banks are connected to the operand collectors via a crossbar.

**Warp Control Block.** A key structure in LTRF design is the *Warp Control Block (WCB)*, shown in Figure 7. The purpose of the WCB is to maintain metadata for each warp required for controlling the register prefetching process and finding the position of the architectural registers in the register cache. To this end, WCB is composed of the *register cache address table*, the *working-set bit-vector*, and the *liveness bit-vector*. The register cache address table is a 256-entry table per warp that keeps the register file cache bank number for each warp's architectural registers. The register cache address table has as many entries as the maximum number of architectural registers allocated to a warp. All cached registers of a warp have the same offset in all register file cache banks. Thus, for each register, the table only needs to keep track of the $\lceil \log_2 \#Registers_per_Interval \rceil$-bit (e.g., 4-bit in Figure 7) index of the register file cache *bank number* where that register is located. WCB also contains one $\lceil \log_2 \#Active_Warps \rceil$-bit (e.g., 3-bit in Figure 7) entry to track the offset of that warp's registers inside the banks (called *warp-offset address*). The *working-set bit-vector* holds a valid bit for each register to indicate whether it has already been prefetched during the PREFETCH phase. Since most of the instructions have two read operands, we provide two read ports for each register cache address table. Any instruction that operates on more than two operands must fetch the register file cache addresses of all operands over multiple cycles.

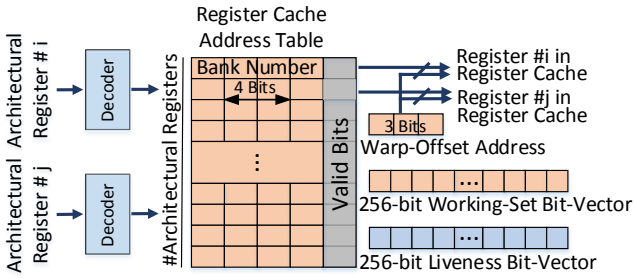


**Figure 7.** Warp control block.

In LTRF+, which considers the liveness of operands, each warp maintains a *liveness bit-vector* that keeps track of the liveness of

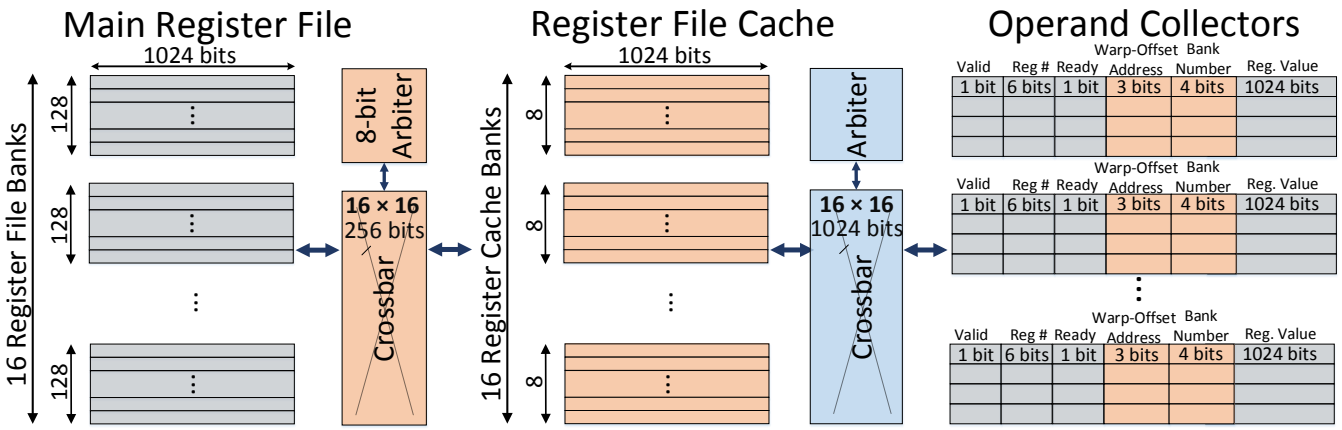


**Figure 5.** LTRF architecture. Figure assumes 8 active warps, 256 architectural registers per warp, 16 register file (cache) banks, and 16 operand collectors.

each architectural register in the WCB, as depicted in Figure 7. This vector is initially cleared (i.e., all registers are marked as dead) when the warp starts execution, and it is updated as the warp executes (§ 3.2).

**Operand Collector Modifications.** We augment each operand collector (Figure 5, right) with $\lceil \log_2 \#Registers_per_Interval \rceil$-bit (e.g., 4 bits in the figure) *bank number* and $\lceil \log_2 \#Active_Warps \rceil$-bit (e.g., 3 bits in the figure) *warp-offset address* to determine the location of each architectural register in the register file cache.

**Register File Cache Access.** As multiple warps may still try to access the same bank at any given cycle, we use an arbiter, as in conventional GPU register files, to arbitrate between accesses to register file cache banks and to resolve bank conflicts. When an operand collector is allocated to a warp, it probes the register cache address table in the corresponding WCB to get the locations of registers inside the register cache. After reading the registers' locations, the operand collector participates in the arbitration phase to 1) resolve bank conflicts and 2) access the register file cache to read the operands.

### 4.2 Software-Triggered Prefetch Mechanism

**Executing PREFETCH Operations.** When a warp reaches a PREFETCH operation, the GPU must load the warp's registers into the register file cache as indicated by the PREFETCH bit-vector. Initially, the PREFETCH bit-vector is decoded into a list of indices (IDs) of registers that need to be loaded. Once the register indices are identified, they must be allocated space in the register file cache, and the warp's register cache address table in the WCB must be properly filled. After allocating register file cache space, the registers can be read from the main register file to fill the register cache. When a register is prefetched completely, the corresponding valid bit in the WCB is set. After all registers indicated by the PREFETCH bit-vector are prefetched, the warp becomes ready to execute, and all subsequent register accesses of that warp are served from the register file cache. In LTRF+, whenever a warp performs a PREFETCH operation, it queries the liveness bit-vector and prefetches *only* the registers that are marked as live. For dead registers, it is sufficient to allocate the register file cache space, without fetching data.

**Register File Cache Space Allocation.** Every cached register in a register-interval must be assigned a place in the register file cache. In our design, this mechanism is equivalent to allocating one register file cache bank for each cached register as we interleave the registers of a single warp across banks to minimize register file cache bank conflicts. We employ the *Address Allocation Unit*, depicted in Figure 8, for *each warp* to implement this mechanism. The Address Allocation Unit is composed of two queues: the *unused* queue keeps track of free banks, while the *occupied* queue keeps track of allocated banks. Initially, the unused queue is full, and the occupied queue is empty. On an allocation, we allocate the head of the unused queue to the new register and move that entry to the occupied queue. On a deallocation, we move the deallocated register entry back to the unused queue. The same mechanism is used to allocate warp-offset addresses to warps. There, we use a *global* Address Allocation Unit that is shared by all warps.

**Interconnect.** We use an $\#Active_Warps$-bit arbiter (e.g., 8-bit arbiter in Figure 5, left) to arbitrate among warps to fill the register file cache. Registers are loaded into the register file cache from the main register file via a crossbar network. In order to design this crossbar, we calculate the number of accesses to the main register file in LTRF and the baseline architecture. Our experimental results show that LTRF reduces the number of accesses to the main register file by 4×-6× (as most of the accesses are serviced through the register file cache) and the bandwidth of the 1024-bit crossbar in the baseline architecture *without* register caching [10, 40] is utilized by up to 85%. As a result, we can reduce the bandwidth of the main register file crossbar by 4× without hurting performance. On the downside, the narrower crossbar would exhibit a traversal latency 4× larger (from 1 cycle to 4 cycles) than a wide crossbar in typical scenarios, and far larger latency when the crossbar is saturated and queuing effects become dominant. However, due to the latency-tolerant nature of our design and the fact that register file access latencies are dominated by bank access times rather than crossbar traversals, we find that the longer traversal latency of a narrower crossbar is *not* a significant performance issue. Because warp registers are interleaved across the register cache banks, LTRF improves the *parallelism of accesses in the interconnect*.

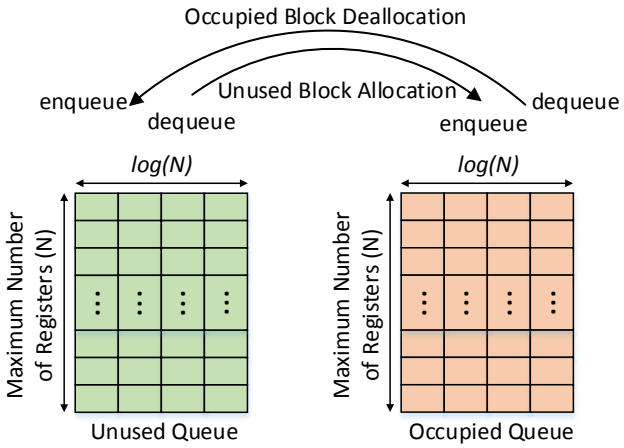


**Figure 8.** Address allocation unit.

**Warp Stall.** A warp that is stalled becomes inactive and loses its slots in the register file cache. In that case, it must perform three steps. First, it writes back its *live* registers to the main register file. Second, it releases its register file cache slots. Third, it clears all the valid bits in the register cache address table of the WCB. Whenever a warp goes from being inactive to active, and it finds itself in the middle of a register-interval, it must refetch all its specified registers in its working-set bit-vector that are still live from the main register file. This is done using the warp's working-set and liveness bit-vectors in WCB.

### 4.3 Overheads

**Code Size.** LTRF increases the average code size by 7% if only PREFETCH bit-vectors are inserted into the code and 9% if the bit-vectors are accompanied by prefetch instructions. Note that, in order to be able to only insert the bit-vectors into the code, the ISA has to be redesigned and an extra bit has to be embedded into *all* instructions, as explained in §3.2. To measure the effect of the increased code size on the GPU performance, we execute the original and the modified programs on the baseline architecture using GPGPU-Sim [10]. Our experimental results show that the larger code size results in 0.2% average (up to 1%) performance degradation, which is negligible.

**Storage Cost.** LTRF requires a WCB for every warp, shown in Figure 7. Each WCB contains one 5-bit entry per architectural register, 3-bit for the warp-offset address, and working-set and liveness bit-vectors, each with one bit per register. The total storage overhead of the WCB for each SM in an example modern architecture, which supports 64 warps with 256 registers per warp, is 114880 bits ($64 \times (256 \times 5 + 3 + 256 + 256)$), around 5% of the area consumed by the 256KB baseline register file.

**Latency Overhead.** According to our analysis with CACTI [51], the WCB can be accessed within one extra clock cycle. Hence, it adds negligible performance overhead in accessing the registers.

**Area/Power Cost.** In order to measure the area and power overheads of LTRF, we functionally model all the added components (i.e., WCB, the additional crossbar, address allocation units, the arbiter, additional entries in the operand collectors, and register file cache) in GPU-Wattch [40]. In total, LTRF occupies 16% more area than our baseline GPU register file (i.e., Configuration #1 in Table 2) using the same main register file size and technology. In terms of power consumption, despite the added structures, LTRF consumes 23% less power compared to the baseline register file. LTRF's improvement in power consumption is due to reducing the number of accesses to the main register file by 4×-6×.

## 5 Methodology

**Simulation.** We evaluate our techniques using the GPGPU-Sim V3.2.2 [10] cycle-level simulator for GPUs. Table 3 provides the details of our baseline GPU configuration. We model our baseline after an NVIDIA Maxwell-like architecture [56]. We modify the microarchitecture of the conventional register file in GPGPU-Sim to implement the LTRF microarchitecture depicted in Figure 5.

| | |
|---|---|
| Number of SMs | 24 |
| Core clock | 1137 MHz |
| Scheduler | Two-level [19, 53] |
| Number of warps per SM | 64 |
| Register file size | 256KB per SM (65536 registers) |
| Register file cache size | 16KB per SM (4096 registers) |
| Shared memory size | 64KB per SM |
| L1D Cache | 4-way, 16KB, 128B line |
| L1I Cache | 4-way, 2KB, 128B line |
| LLC | 8-way, 2MB, 128B line |
| Memory Model | 8 GDDR5 MCs, FR-FCFS [61, 88], 2700 MHz |
| GDDR5 Timing (in nanoseconds) | $t_{CL}$=12, $t_{RP}$=12, $t_{RC}$=40, $t_{RAS}$=28, $t_{RCD}$=12, $t_{RRD}$=6 |
| Number of active warps | 8 per SM |
| Number of registers in a register-interval | 16 |

**Table 3.** Simulated system configuration.

We use the compiler in GPGPU-Sim to implement our software prefetching mechanism for registers. To this end, we process the CFG of the register-allocated PTX code to form the register-intervals, using Algorithms 1 and 2, and insert PREFETCH bit-vectors at the start of each register-interval.

**Benchmarks.** We run 35 benchmarks from CUDA SDK [10], Rodinia [14], and Parboil [72] benchmark suites and classify them into two groups, *register-sensitive* and *register-insensitive*, based on whether or not the register file limits the achievable TLP. We randomly select nine workloads from the register-sensitive group, and five workloads from the register-insensitive one.

**Comparison Points.** We evaluate (1) a baseline (BL) architecture that models a GPU with a conventional non-cached register file. To provide a fair comparison of this baseline to other register file cache based designs, we add the amount of space dedicated for the register file cache in LTRF (16KB) to the main register file capacity in the BL architecture, (2) a design with a 16KB register file cache (RFC) *without* any prefetching mechanisms, similar to the architecture proposed in [19]. (3) LTRF, with a 16KB register file cache. (4) LTRF+, an enhanced version of LTRF which also considers operand-liveness information (§ 3.2). (5) an *Ideal* register file architecture that allows us to increase the register file capacity to any size (i.e., 8× in our evaluations) with *no latency overhead*.

**Design Points.** We increase the register file size from 256KB to 2MB by using the register file configurations #6 and #7 from Table 2. Configuration #6 allows us to increase the register file size by 8× while keeping the power consumption almost unchanged. Configuration #7, on the other hand, results in less power/area consumption compared to the baseline SRAM-based 256KB register file. We use these design points as realistic baselines for our performance analysis.

**Performance Metrics.** We use IPC as the performance metric to evaluate different register file designs. We evaluate our compiler algorithms by measuring the size of the generated register-intervals.

## 6 Evaluation

We present the effectiveness of five different mechanisms: BL, RFC, LTRF, LTRF+, and Ideal. § 6.1 shows the overall effect of LTRF on GPU performance. § 6.2 evaluates the register file power consumption. § 6.3 analyzes the effectiveness of LTRF at tolerating the latency of the main register file. § 6.4 provides sensitivity analysis on the size of the register file cache. § 6.5 analyzes the number of instructions in register-intervals. § 6.6 provides a comparison between LTRF and other software-managed register caching schemes.

### 6.1 Overall Effect on GPU Performance

To evaluate the effect of larger register files on GPU performance, we increase the register file size from 256KB to 2MB by using the register file configurations #6 and #7 from Table 2. Figure 9 compares the normalized IPC of BL, RFC, LTRF, and LTRF+ designs when used on top of these two configurations. In this figure, we normalize the IPC results to the IPC results of the baseline architecture of configuration #1 in Table 2, without any register caching, with one modification: we add the register file cache capacity (i.e., 16KB) used in the other mechanisms to the 256KB register file size of configuration #1. Ideal bars show the IPC of an idealized version of configuration #1 with 8× the register file capacity but *no* increase in latency (i.e., access latency remains constant after increasing register file size by 8×). We make three major observations. First, LTRF provides almost the same IPC performance as the Ideal design when we employ configuration #6. LTRF improves IPC by 32%, on average. The IPC improvement of LTRF is due to two main reasons. 1) The larger register file enables both more registers per thread and more warps executing in parallel. 2) LTRF effectively tolerates the higher access latency of configuration #6. Second, for the register-insensitive workloads that do *not* benefit from a larger register file

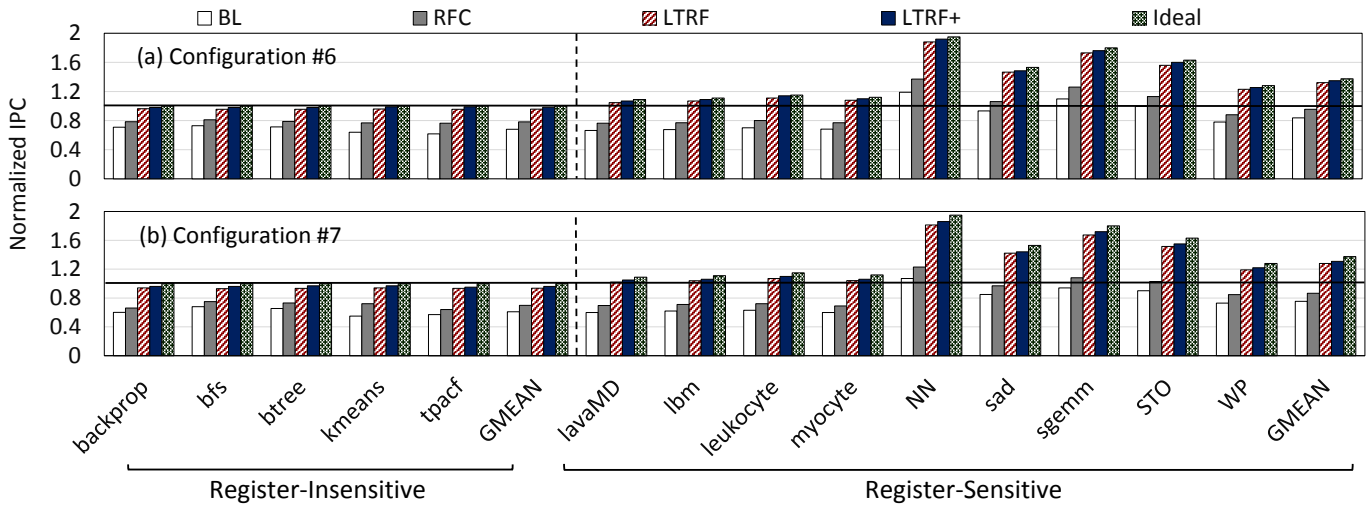


**Figure 9.** IPC of BL, RFC, LTRF, LTRF+, and Ideal using the main register file configurations #6 and #7 from Table 2, normalized to the baseline architecture of configuration #1 with 16KB additional register file capacity.

(e.g., `btree` and `kmeans`), the performance overhead of increasing the register file size is minimal if we use LTRF and LTRF+ as opposed to RFC. Third, LTRF and LTRF+ effectively enable the use of configuration #7, which reduces the register file area by 75%.[4] For this configuration, LTRF and LTRF+ improve performance by 28% and 31% over the baseline, on average, respectively. We conclude that LTRF enables a high-capacity and high-latency main register file while providing high performance.

### 6.2 Register File Power Consumption

To evaluate the effect of each design on register file power consumption, we measure the register file power consumption using configuration #7 in Table 2. Figure 10 compares the normalized register file power consumption of RFC, LTRF, and LTRF+ designs when used on top of configuration #7. We normalize the power consumption results to the power consumption of the baseline architecture of configuration #1 from Table 2, without any register caching.

We make two major observations. First, compared to the baseline, LTRF+ consumes the least amount of power. On average, LTRF+ reduces the register file power consumption by 46.1% while RFC and LTRF reduce the register file power consumption by 35.1% and 35.4%, respectively. Second, the register file of LTRF consumes almost the same amount of power of that of RFC because the extra storage components (e.g., WCB) offset the power saving from LTRF's lower number of main register file accesses.

### 6.3 Effect of LTRF on Register File Access Latency

To show the effectiveness of LTRF at tolerating register file access latency, we define a new metric: the *maximum tolerable register file access latency*. This is the relative latency[5] of the main register file that leads to at most 5% performance (IPC) loss for each workload we examine. Note that this metric is different for each design, depending on the latency tolerance of the design. We increase the main register file access latency while keeping the main register file size constant. Figure 11 compares the maximum tolerable register file access latency of different designs for various benchmarks.

We make three major observations. First, the maximum tolerable register file access latency for LTRF is 5.3×, on average. This result indicates that LTRF can 1) effectively bring the registers to the register file cache before they are accessed and 2) hide the latency of the register access to the main register file by executing other active

[4] This extra die area is freed up due to the use of LTRF and LTRF+, and it can naturally be used for other on-chip components such as the L1 and the L2 data caches (although we do not evaluate this use of the extra area).

[5] Relative to the baseline of configuration #1 with 16KB additional register file capacity

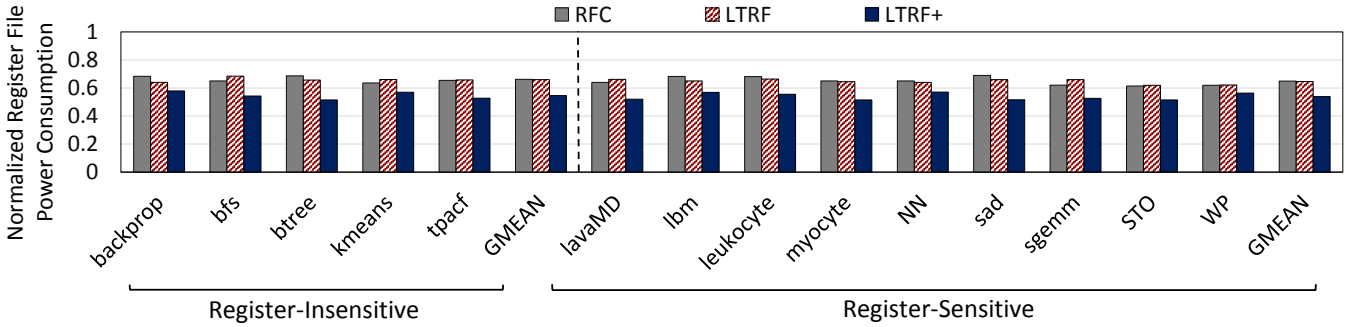


**Figure 10.** Register file power consumption of RFC, LTRF, and LTRF+ using the main register file configuration #7 from Table 2, normalized to the baseline architecture of configuration #1.

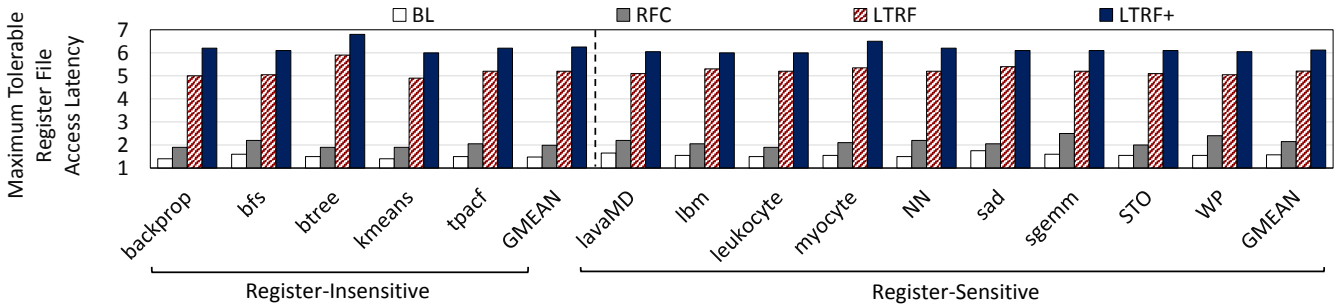


**Figure 11.** Maximum tolerable register file access latency of different designs.

warps. Second, the maximum tolerable register file access latency for LTRF+, which is aware of operand liveness information, is 6.2×, on average, indicating that using liveness information to manage register caching and prefetching improves latency tolerance. Third, the maximum tolerable register file access latency for RFC is 2.1×, on average, which shows that the register file cache hit rate in the RFC design is *not* large enough to hide main register file access latencies that are greater than 2.1×.

We also evaluate the maximum tolerable register file access latency for different designs by allowing 1% and 10% performance loss instead of 5% performance loss. With 1% allowable performance loss, the maximum tolerable register file access latencies for RFC, LTRF, and LTRF+ are 1.4×, 2.8× and 3.5× higher than the baseline, respectively. With 10% allowable performance loss, the maximum tolerable register file access latencies for RFC, LTRF, and LTRF+ are 2.9×, 6.5× and 7.9× higher than the baseline, respectively.

We conclude that LTRF and LTRF+ are able to tolerate long main register file access latencies. Thus, they can enable aggressive optimizations that increase register file capacity in exchange for higher access latency.

### 6.4 Sensitivity to Register File Cache Size

We explore the effect of the register file cache size on performance in two ways: (1) varying the number of registers allowed in each register-interval (default is 16), (2) varying the number of active warps that are allocated storage space in the register cache. Figure 12 reports the average IPC when we vary the number of registers allowed in each register-interval. We make two observations. First, when the number of registers allowed in each register-interval is 8, the effectiveness of LTRF degrades significantly, as the main register file access latency increases, This is mainly because a small number of registers results in a small register-interval size. Hence, PREFETCH operations become more frequent, and hiding their latency becomes more difficult, especially for slow main register files. Second, increasing the number of registers allowed in each register-interval does *not* necessarily translate to better performance for LTRF. This is mainly because more registers result in more main register file bank conflicts during the PREFETCH operation, increasing prefetch latency. Therefore, larger register-interval sizes may not always be enough to hide larger prefetch latencies.

Figure 13 illustrates LTRF performance sensitivity to the number of warps that have dedicated register file cache space, while keeping the dedicated space per warp constant. We make two observations. First, as the number of active warps increases from 4 to 8, IPC improves by 36.9% for the slowest main register file. Second, increasing the number of active warps by more than 8 does *not*

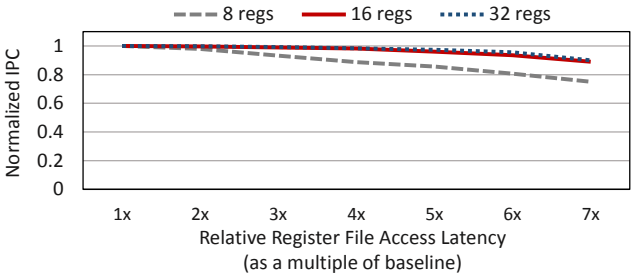


**Figure 12.** Normalized IPC using LTRF with various main register file access latencies and number of allowed registers in each register-interval.

have a significant impact on LTRF performance. We conclude that 8 active warps, which is the default configuration in LTRF, seems enough. Hence, LTRF does *not* impose significant performance cost by limiting the number of active warps.

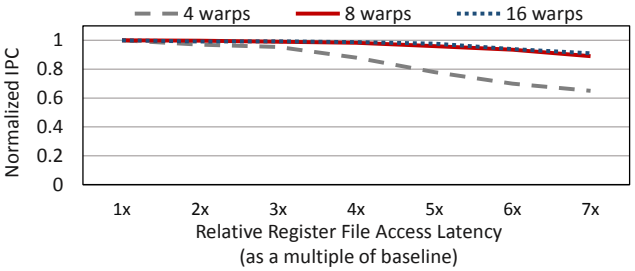


**Figure 13.** Normalized IPC using LTRF with various main register file access latencies and number of active warps.

We conclude that, by making the performance impact of a slower register file more tolerable, LTRF enables a large design space to architects, where tradeoffs between power, area, and latency of the register file can be explored more freely to optimize system-level goals.

### 6.5 Register-Interval Length

As explained in § 3, register-intervals should be as long as possible to minimize the number of PREFETCH operations. We measured both the real and the optimal register-interval lengths. The *real register-interval length* is the number of dynamic instructions within each register-interval. The *optimal register-interval length* is the number of consecutive dynamic instructions in a kernel's execution trace that consume at most the maximum number of allowed registers

in the register cache. In other words, the optimal length exposes the limitations caused by the control-flow constraints imposed on register-intervals. Table 4 reports the average, minimum, and maximum lengths of the real and optimal register-intervals. We make two observations. First, the real register-interval length is 89% of the optimal register-interval length, on average. Second, the minimum and maximum lengths of real register-intervals are 78% and 85% of the ones in optimal register-intervals, respectively. We conclude that the control flow constraints in creating the register-intervals do *not* greatly limit the register-interval length.

| Register-Interval Length | Average | Minimum | Maximum |
|---|---|---|---|
| Real | 31.2 | 7 | 45 |
| Optimal | 34.7 | 9 | 53 |

**Table 4.** The average, minimum, and maximum lengths of real and optimal register-intervals, in terms of dynamic instructions, for 35 workloads in CUDA SDK [10], Rodinia [14], and Parboil [72] benchmark suites.

### 6.6 LTRF vs. SW-Managed Hierarchical Register Files

To distinguish the benefits of our key ideas from other software-based approaches, we evaluate the maximum tolerable register file access latency of two additional designs: 1) a software-managed hierarchical register file (SHRF) similar to [20] and 2) a version of LTRF that performs PREFETCH operations at the end of *strands* [20], rather than register-intervals. SHRF [20] aims to reduce the number of background register swap operations between the main register file and the register file cache to provide energy efficiency and uses traditional register allocation/spilling techniques. SHRF uses strands, which are more constrained CFG subgraphs than register-intervals, since long/variable-latency operations (e.g., cache misses) and backward branches are disallowed within a strand to guarantee that the warp does *not* get descheduled until the end of the strand [20].

Figure 14 reports the normalized IPC, averaged across our workloads (see § 5), as the main register file access latency increases.

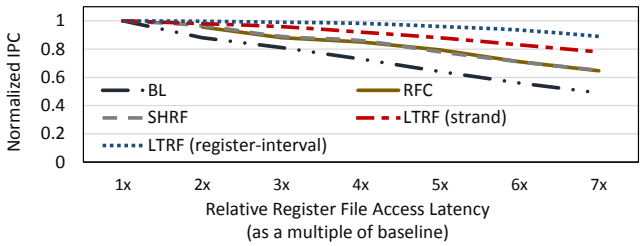


**Figure 14.** Normalized IPC using BL, RFC, SHRF, LTRF (strand), and LTRF (register-interval) with various register file access latencies.

We make two observations. First, SHRF performs similarly to RFC and can tolerate latencies by up to 2× the baseline latency. Second, LTRF can tolerate only 3× higher register file latency if it uses strands instead of register-intervals, as opposed to the 5.3× higher main register file latency tolerated by our LTRF design that uses register-intervals. LTRF performs better using register-intervals because strands' CFG subgraphs are more constrained, and typically much smaller than the CFG subgraphs of register-intervals, increasing the number of PREFETCH operations, register writebacks, and register re-fetches. In particular, while the length of a register-interval is usually limited by the size of the register working-set, a strand is typically terminated due to unrelated control flow constraints, and as a result, the strand's register working-set is often smaller than the available register file cache space. We conclude that using register-intervals to place the PREFETCH operations is essential for LTRF performance.

## 7 Related Work

To our knowledge, this paper is the first to design a latency-tolerant register file architecture for GPUs by (1) prefetching the entire register working set of a warp from the main register file to the register file cache using the notion of register-intervals, and (2) overlapping the prefetch latency with the execution of other active warps. LTRF opens a window for many optimizations in the main register file that greatly increase the effective capacity at the expense of higher access latency. We have already compared LTRF extensively to various hardware and software register caching proposals for GPUs [19, 20] in § 6. In this section, we describe other related work in register file caching and register file scalability.

**Register File Caching.** Few works have explored hardware- and software-managed hierarchical register files for GPUs [19, 20]. These works focus on other objectives, such as energy efficiency, rather than latency-tolerance, and expose the higher latencies of slow register files to the execution. Regless [36] is a concurrent work that slices the computation graph into regions and allocates operand storage for the regions to replace the register file with a small operand staging unit. However, Regless targets power reduction rather than latency tolerance as the main objective.

Register file caching and hierarchical register files have been widely investigated for CPU architectures. Most of those works focus on superscalar or VLIW processors [11, 13, 16, 54, 58, 70, 86, 87]. Such architectures are often able to hide the larger access latency of the slower register file levels via instruction level parallelism. As a result, the main focus of this line of work has been on efficient ways of integrating hierarchical register files into deep out-of-order pipelines and orchestrating the interactions between rename/issue mechanisms and register movements among different levels [11, 58]. However, these techniques are usually not applicable to GPUs as GPUs have limited support for instruction-level parallelism. Another line of work focuses on software-managed hierarchical files with different ISA-visible register banks that have different sizes and speeds [15, 31, 32, 73] where the compiler orchestrates register placement and movement. The CRAY-1 system [63] is an example architecture that implements a compiler-controlled hierarchical register file where software instructions explicitly manage the register movement between the two levels. Such techniques are suitable mainly for VLIW/vector processors and are not effective when used with GPUs where dynamic thread interleavings are unknown at compile-time as the GPU compiler is not able to schedule register movements to overlap them with the execution of other threads.

**Register File Scalability.** There are many techniques that improve the scalability or efficiency of the register files. These techniques employ dimming and power-gating [2, 23], compression [39], new memory technologies [3, 26, 27, 28, 43, 45, 46, 48, 80, 85], and virtualization [25, 78]. All these techniques likely cause an increase in register file access latency. LTRF can be synergistically combined

with these techniques and can enable them to tolerate the long register file access latencies. Hence, we believe LTRF is a substrate that enables optimizations in GPU register files, which might otherwise not always be desirable, efficient, or high performance.

## 8 Conclusion

We propose LTRF, a new *latency-tolerant* hierarchical register file design for GPUs. The key mechanism of LTRF is a near-perfect register prefetching scheme that divides the application control flow graph into register-intervals and brings the entire register working set of a warp from the main register file to the register cache at the beginning of each register-interval. As a result, a warp experiences the fast register cache access latency, rather than the long access latency of the large main register file. An example evaluation result shows that LTRF enables us to implement the main register file with emerging high-density high-latency memory technologies, enabling 8× larger register file capacity and improving overall GPU performance by 31% while reducing register file power consumption by 46%. We believe that LTRF paves the way for many power/area optimization techniques in the main register file that likely increase the register access latency. We conclude that, by making the performance impact of a slower register file more tolerable, LTRF enables a large design space to architects, where tradeoffs between power, area, and latency of the register file can be explored more freely to optimize system-level goals.

## Acknowledgements

We thank the anonymous reviewers, and members of HPCAN-Sharif, PARSA-EPFL, SAFARI-ETH, and SAFARI-CMU, in particular Juan Gómez-Luna, Ali Hajiabadi, and Mark Johnathon Sutherland, for their feedback. We acknowledge the support of our industrial partners, especially Google, Intel, Microsoft, NVIDIA, and VMware.

## References

[1] "LTRF Register-Interval-Algorithm," https://github.com/CMU-SAFARI/Register-Interval.
[2] M. Abdel-Majeed and M. Annavaram, "Warped register file: A power efficient register file for GPGPUs," in *HPCA*, 2013.
[3] M. Abdel-Majeed, A. Shafaei, H. Jeon, M. Pedram, and M. Annavaram, "Pilot Register File: Energy efficient partitioned register file for GPUs," in *HPCA*, 2017.
[4] A. Annunziata, M. Gaidis, L. Thomas, C. Chien, C. Hung, P. Chevalier, E. O'Sullivan, J. Hummel, E. Joseph, Y. Zhu *et al.*, "Racetrack memory cell array with integrated magnetic tunnel junction readout," in *IEDM*, 2011.
[5] C. Augustine, A. Raychowdhury, B. Behin-Aein, S. Srinivasan, J. Tschanz, V. K. De, and K. Roy, "Numerical analysis of domain wall propagation for dense memory arrays," in *IEDM*, 2011.
[6] R. Ausavarungnirun, S. Ghose, O. Kayiran, G. H. Loh, C. R. Das, M. T. Kandemir, and O. Mutlu, "Exploiting inter-warp heterogeneity to improve gpgpu performance," in *PACT*, 2015.
[7] A. Bakhoda, J. Kim, and T. M. Aamodt, "On-chip network design considerations for compute accelerators," in *PACT*, 2010.
[8] A. Bakhoda, J. Kim, and T. M. Aamodt, "Throughput-effective on-chip networks for manycore accelerators," in *MICRO*, 2010.
[9] A. Bakhoda, J. Kim, and T. M. Aamodt, "Designing on-chip networks for throughput accelerators," *in ACM TACO*, 2013.
[10] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *ISPASS*, 2009.
[11] R. Balasubramonian, S. Dwarkadas, and D. H. Albonesi, "Reducing the complexity of the register file in dynamic superscalar processors," in *MICRO*, 2001.
[12] K. K. Bhuwalka, S. Sedlmaier, A. K. Ludsteck, C. Tolksdorf, J. Schulze, and I. Eisele, "Vertical tunnel field-effect transistor," *in IEEE TED*, 2004.
[13] E. Borch, E. Tune, S. Manne, and J. Emer, "Loose loops sink chips," in *HPCA*, 2002.
[14] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *IISWC*, 2009.
[15] K. D. Cooper and T. J. Harvey, "Compiler-controlled memory," in *ASPLOS*, 1998.
[16] J. L. Cruz, A. Gonzalez, M. Valero, and N. P. Topham, "Multiple-banked register file architectures," in *ISCA*, 2000.
[17] X. Dong, C. Xu, Y. Xie, and N. P. Jouppi, "Nvsim: A circuit-level performance, energy, and area model for emerging nonvolatile memory," *in IEEE TCAD*, 2012.
[18] S. Fukami, T. Suzuki, K. Nagahara, N. Ohshima, Y. Ozaki, S. Saito, R. Nebashi, N. Sakimura, H. Honjo, K. Mori *et al.*, "Low-current perpendicular domain wall motion cell for scalable high-speed mram," in *VLSIT*, 2009.
[19] 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.
[20] M. Gebhart, S. W. Keckler, and W. J. Dally, "A compile-time managed multi-level register file hierarchy," in *MICRO*, 2011.
[21] 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 *MICRO*, 2012.
[22] M. S. Hecht, *Flow analysis of computer programs.* Elsevier Science Inc., 1977.
[23] C.-C. Hsiao, S.-L. Chu, and C.-C. Hsieh, "An adaptive thread scheduling mechanism with low-power register file for mobile GPUs," *in IEEE TMM*, 2014.
[24] H. Jang, J. Kim, P. Gratz, K. H. Yum, and E. J. Kim, "Bandwidth-efficient on-chip interconnect designs for GPGPUs," in *DAC*, 2015.
[25] H. Jeon, G. S. Ravi, N. S. Kim, and M. Annavaram, "GPU register file virtualization," in *MICRO*, 2015.
[26] N. Jing, L. Jiang, T. Zhang, C. Li, F. Fan, and X. Liang, "Energy-Efficient eDRAM-Based On-Chip Storage Architecture for GPGPUs," *in IEEE TC*, 2016.
[27] N. Jing, H. Liu, Y. Lu, and X. Liang, "Compiler assisted dynamic register file in GPGPU," in *ISLPED*, 2013.
[28] N. Jing, Y. Shen, Y. Lu, S. Ganapathy, Z. Mao, M. Guo, R. Canal Corretger, and X. Liang, "An energy-efficient and scalable eDRAM-based register file architecture for GPGPU," in *ISCA*, 2013.
[29] A. Jog, O. Kayiran, N. Chidambaram Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "OWL: cooperative thread array aware scheduling techniques for improving GPGPU performance," *in ASPLOS*, 2013.
[30] A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "Orchestrated scheduling and prefetching for GPGPUs," *in ISCA*, 2013.
[31] T. M. Jones, M. F. P. O'Boyle, J. Abella, A. González, and O. Ergin, "Energy-efficient register caching with compiler assistance," *in ACM TACO*, 2009.
[32] U. J. Kapasi, W. J. Dally, S. Rixner, J. D. Owens, and B. Khailany, "The imagine stream processor," in *ICCD*, 2002.
[33] O. Kayıran, A. Jog, M. T. Kandemir, and C. R. Das, "Neither more nor less: optimizing thread-level parallelism for GPGPUs," in *PACT*, 2013.
[34] O. Kayiran, N. C. Nachiappan, A. Jog, R. Ausavarungnirun, M. T. Kandemir, G. H. Loh, O. Mutlu, and C. R. Das, "Managing GPU concurrency in heterogeneous architectures," in *MICRO*, 2014.
[35] J. Kim, J. Balfour, and W. Dally, "Flattened butterfly topology for on-chip networks," in *MICRO*, 2007.
[36] J. Kloosterman, J. Beaumont, D. A. Jamshidi, J. Bailey, T. Mudge, and S. Mahlke, "Regless: Just-in-time operand staging for GPUs," in *MICRO*, 2017.
[37] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *CGO*, 2004.
[38] J. Lee, N. B. Lakshminarayana, H. Kim, and R. Vuduc, "Many-thread aware prefetching mechanisms for GPGPU applications," in *MICRO*, 2010.
[39] S. Lee, K. Kim, G. Koo, H. Jeon, W. W. Ro, and M. Annavaram, "Warped-Compression: Enabling power efficient GPUs through register compression," in *ISCA*, 2015.
[40] J. Leng, T. Hetherington, A. Eltantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "GPUWattch: Enabling energy optimizations in GPGPUs," in *ISCA*, 2013.
[41] E. Lewis, D. Petit, L. O'Brien, A. Fernandez-Pacheco, J. Sampaio, A. Jausovec, H. Zeng, D. Read, and R. Cowburn, "Fast domain wall motion in magnetic comb structures," *in Nature Materials*, 2010.
[42] C. Li, S. L. Song, H. Dai, A. Sidelnik, S. K. S. Hari, and H. Zhou, "Locality-driven dynamic GPU cache bypassing," in *ICS*, 2015.
[43] Z. Li, J. Tan, and X. Fu, "Hybrid CMOS-TFET based register files for energy-efficient GPGPUs," in *ISQED*, 2013.
[44] J. E. Lindholm, M. Y. Siu, S. S. Moy, S. Liu, and J. R. Nickolls, "Simulating multiported memories using lower port count memories," 2008, US Patent 7,339,592.
[45] X. Liu, Y. Li, Y. Zhang, A. K. Jones, and Y. Chen, "STD-TLB: A STT-RAM-based dynamically-configurable translation lookaside buffer for GPU architectures," in *ASP-DAC*, 2014.
[46] 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.
[47] A. Magni, C. Dubach, and M. F. P. O'Boyle, "A large-scale cross-architecture evaluation of thread-coarsening," in *SC*, 2013.
[48] 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.
[49] A. Mirhosseini, M. Sadrosadati, B. Soltani, H. Sarbazi-Azad, and T. F. Wenisch, "BiNoCHS: Bimodal network-on-chip for CPU-GPU heterogeneous systems," in *NOCS*, 2017.
[50] S. Mookerjea and S. Datta, "Comparative study of si, ge and inas based steep subthreshold slope tunnel transistors for 0.25 v supply voltage logic applications," in *Device Research Conference*, 2008.

[51] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "CACTI 6.0: A tool to model large caches," HP Laboratories, Tech. Rep., 2009.
[52] G. S. Murthy, M. Ravishankar, M. M. Baskaran, and P. Sadayappan, "Optimal loop unrolling for GPGPU programs," in *IPDPS*, 2010.
[53] 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 *MICRO*, 2011.
[54] P. R. Nuth and W. J. Dally, "The named-state register file: Implementation and performance," in *HPCA*, 1995.
[55] Nvidia, "C programming guide V6. 5. 2014," *San Jose California: Nvidia*.
[56] Nvidia, "White paper: NVIDIA GeForce GTX 980," Nvidia, Tech. Rep.
[57] Nvidia, "White paper: NVIDIA Tesla P100," Nvidia, Tech. Rep.
[58] D. W. Oehmke, N. L. Binkert, T. Mudge, and S. K. Reinhardt, "How to fake 1000 registers," in *MICRO*, 2005.
[59] S. S. Parkin, M. Hayashi, and L. Thomas, "Magnetic domain-wall racetrack memory," *in Science*, 2008.
[60] W. M. Reddick and G. A. Amaratunga, "Silicon surface tunnel transistor," *Applied Physics Letters*, 1995.
[61] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory access scheduling," in *ISCA*, 2000.
[62] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-conscious wavefront scheduling," in *MICRO*, 2012.
[63] R. M. Russell, "The CRAY-1 computer system," *Commun. ACM*, 1978.
[64] M. Sadrosadati, A. Mirhosseini, S. Roozkhosh, H. Bakhishi, and H. Sarbazi-Azad, "Effective cache bank placement for GPUs," in *DATE*.
[65] M. H. Samavatian, H. Abbasitabar, M. Arjomand, and H. Sarbazi-Azad, "An efficient STT-RAM last level cache architecture for GPUs," in *DAC*, 2014.
[66] M. H. Samavatian, M. Arjomand, R. Bashizade, and H. Sarbazi-Azad, "Architecting the last-level cache for GPUs using STT-RAM technology," *in ACM TODAES*, 2015.
[67] A. Sethia, G. Dasika, M. Samadi, and S. Mahlke, "APOGEE: Adaptive prefetching on GPUs for energy efficiency," in *PACT*, 2013.
[68] A. Sethia and S. Mahlke, "Equalizer: Dynamic tuning of gpu resources for efficient execution," in *MICRO*, 2014.
[69] M. Sharad, R. Venkatesan, A. Raghunathan, and K. Roy, "Multi-level magnetic RAM using domain wall shift for energy-efficient, high-density caches," in *ISLPED*, 2013.
[70] R. Shioya, K. Horio, M. Goshima, and S. Sakai, "Register cache system not for latency reduction purpose," in *MICRO*, 2010.
[71] J. Singh, K. Ramakrishnan, S. Mookerjea, S. Datta, N. Vijaykrishnan, and D. Pradhan, "A novel si-tunnel FET based SRAM design for ultra low-power 0.3V VDD applications," in *ASP-DAC*, 2010.
[72] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," Center for Reliable and High-Performance Computing, UIUC, Tech. Rep., 2012.
[73] J. A. Swensen and Y. N. Patt, "Hierarchical registers for scientific computers," in *ICS*, 1988.
[74] L. Thomas, R. Moriya, C. Rettner, and S. S. Parkin, "Dynamics of magnetic domain walls under their own inertia," *in Science*, 2010.
[75] Y. Tian, S. Puthoor, J. L. Greathouse, B. M. Beckmann, and D. A. Jiménez, "Adaptive GPU Cache Bypassing," in *GPGPU*, 2015.
[76] R. Venkatesan, S. G. Ramasubramanian, S. Venkataramani, K. Roy, and A. Raghunathan, "Stag: Spintronic-tape architecture for GPGPU cache hierarchies," in *ISCA*, 2014.
[77] R. Venkatesan, M. Sharad, K. Roy, and A. Raghunathan, "Dwm-tapestri-an energy efficient all-spin cache using domain wall shift based writes," in *DATE*, 2013.
[78] N. Vijaykumar, K. Hsieh, G. Pekhimenko, S. Khan, A. Shrestha, S. Ghose, A. Jog, P. B. Gibbons, and O. Mutlu, "Zorua: A holistic approach to resource virtualization in GPUs," in *MICRO*, 2016.
[79] N. Vijaykumar, G. Pekhimenko, A. Jog, A. Bhowmick, R. Ausavarungnirun, C. Das, M. Kandemir, T. C. Mowry, and O. Mutlu, "A case for core-assisted bottleneck acceleration in GPUs: enabling flexible data compression with assist warps," in *ISCA*, 2015.
[80] J. Wang and Y. Xie, "A write-aware STTRAM-based register file architecture for GPGPU," *in ACM JETC*, 2015.
[81] P.-F. Wang, "Complementary tunneling-FETs (CTFET) in CMOS technology," Ph.D. dissertation, Technische Universität München, Universitätsbibliothek, 2003.
[82] X. Xie, Y. Liang, X. Li, Y. Wu, G. Sun, T. Wang, and D. Fan, "Enabling coordinated register allocation and thread-level parallelism optimization for GPUs," in *MICRO*, 2015.
[83] X. Xie, Y. Liang, G. Sun, and D. Chen, "An efficient compiler framework for cache bypassing on GPUs," in *ICCAD*, 2013.
[84] Y. Yang, P. Xiang, J. Kong, M. Mantor, and H. Zhou, "A unified optimizing compiler framework for different GPGPU architectures," *in ACM TACO*, 2012.
[85] 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.
[86] R. Yung and N. C. Wilhelm, "Caching processor general registers," in *ICCD*, 1995.
[87] H. Zeng and K. Ghose, "Register file caching for energy efficiency," in *ISLPED*, 2006.
[88] W. K. Zuravleff and T. Robinson, "Controller for a synchronous DRAM that maximizes throughput by allowing memory requests and commands to be issued out of order," 1997, US Patent 5,630,096.