CORF: Coalescing Operand Register File for GPUs

Hodjat Asghari Esfeden
University of California, Riverside
Riverside, CA
hodjat.asghari@email.ucr.edu

Farzad Khorasani*
Tesla, Inc.
Palo Alto, CA
fkhorasani@tesla.com

Hyeran Jeon
San Jose State University
San Jose, CA
hyeran.jeon@sjsu.edu

Daniel Wong
University of California, Riverside
Riverside, CA
danwong@ucr.edu

Nael Abu-Ghazaleh
University of California, Riverside
Riverside, CA
nael@cs.ucr.edu

Abstract

The Register File (RF) in GPUs is a critical structure that maintains the state for thousands of threads that support the GPU processing model. The RF organization substantially affects the overall performance and the energy efficiency of a GPU. For example, the frequent accesses to the RF consume a substantial amount of the dynamic energy, and port contention due to limited ports on operand collectors and register file banks affect performance as register operations are serialized. We present CORF, a compiler-assisted Coalescing Operand Register File which performs register coalescing by combining reads to multiple registers required by a single instruction, into a single physical read. To enable register coalescing, CORF utilizes register packing to co-locate narrow-width operands in the same physical register. CORF uses compiler hints to identify which register pairs are commonly accessed together. CORF saves dynamic energy by reducing the number of physical register file accesses, and improves performance by combining read operations, as well as by reducing pressure on the register file. To increase the coalescing opportunities, we re-architect the physical register file to allow coalescing reads across different physical registers that reside in mutually exclusive sub-banks; we call this design CORF++. The compiler analysis for register allocation for CORF++ becomes a form of graph coloring called the bipartite edge frustration problem. CORF++ reduces the dynamic energy of the RF by 17%, and improves IPC by 9%.

*This work was done while the author was at Georgia Tech.

ASPLOS '19, April 13–17, 2019, Providence, RI, USA

ACM ISBN 978-1-4503-6240-5/19/04…$15.00
https://doi.org/10.1145/3297858.3304026

CCS Concepts • **Computer systems organization → Architectures**; **Single instruction, multiple data**; • **Software and its engineering →** *Compilers.*

Keywords register coalescing; GPU; register file; microarchitecture; compiler; graph coloring; register packing;

ACM Reference Format:
Hodjat Asghari Esfeden, Farzad Khorasani, Hyeran Jeon, Daniel Wong, and Nael Abu-Ghazaleh. 2019. CORF: Coalescing Operand Register File for GPUs. In *2019 Architectural Support for Programming Languages and Operating Systems (ASPLOS '19), April 13–17, 2019, Providence, RI, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3297858.3304026

1 Introduction

Over the past decade, GPUs have continued to grow in terms of performance and size. The number of execution units has been steadily increasing, which in turn increases the number of concurrent thread contexts needed to keep these units utilized [24–26, 37, 38, 40, 41, 47]. In order to support fast context switching between large groups of active threads, GPUs invest in large register files to allow each thread to maintain its context. This design enables fine-grained switching between executing groups of threads, which is necessary to hide the latency of data accesses. For example, the Nvidia Volta GPU has 80 streaming multiprocessors each with a 256KB register file (64K registers, each 32-bit wide) for a total of 20MB of register file space. Due to its continuous access, the register file is a critical structure for sustaining performance. The register file is the largest SRAM structure on the die and one of the most power-hungry components on the GPU. In 2013, it was estimated that the register file is responsible for 18% of the total power consumption on a GPU chip [28], a percentage that is likely to have increased as the size of the RF has continued to grow.

In this paper, we seek to improve the performance and energy efficiency of GPU register files by introducing *register coalescing*[1]. Similar to memory coalescing where contiguous memory accesses are combined into a single memory

[1]"Register coalescing" is analogous to memory coalescing where requests are coalesced [8], and distinct from register coalescing in compiler register allocation which is used to eliminate copy instructions [13, 19, 43].

request, register coalescing combines multiple register reads from the same instruction into a single physical register read, provided these registers are stored in the same physical register entry. Specifically, register coalescing opportunities are possible when we use register packing [16, 56], where multiple narrow-width registers are stored into the same physical register. In contrast to register packing, which requires one separate read access for each architectural register read, register coalescing allows combining of read operations to multiple architectural registers that are stored together in the same physical register entry. Register coalescing reduces dynamic access energy, improves register file bandwidth, reduces contention for register file and operand collector ports, and therefore improves overall performance.

We propose a Coalescing Operand Register File (CORF) to take advantage of register coalescing opportunities through a combination of compiler-guided register allocation and coalescing-aware register file organization. The key to increasing register coalescing opportunities is to ensure that *related registers*—registers that show up as source operands in the same instruction—are stored together in the same physical register entry. CORF first identifies exclusive *common pairs* of registers that are most frequently accessed together *within the same instruction*. If both common pair registers are narrow-width and are packed together into the same physical register entry, then accesses to these registers (in the same instruction) can be coalesced. CORF reduce physical register accesses, resulting in ~8.5% reduction in register file dynamic energy, and ~4% increase in IPC.

A limitation of CORF is that each register may only be coalesced exclusively with one other register, which limits the opportunities for coalescing registers that are frequently read with several other registers. To further increase register coalescing opportunities, we present CORF++ which presents a re-architected coalescing-aware register file organization that enables coalescing reads from non-overlapping sub-banks across different physical register entries. Thus, reads to any two registers that reside in non-overlapping sub-banks, even if they reside in different physical register entries, can be coalesced. To maximize the opportunities for coalescing, we introduce a compiler-guided run-time register allocation policy which takes advantage of this reorganization. In particular, we show that the compiler must solve a graph coloring variant called the bipartite edge frustration problem to optimize allocation. Since the problem is NP-hard, we use a heuristic to determine how to allocate the registers effectively. CORF++ is able to substantially improve register coalescing opportunities, leading to a reduction in dynamic register file energy by 17% and an IPC improvement of ~9% over the baseline.

As a secondary contribution, we show that CORF can be combined seamlessly with register file virtualization [21] to further reduce the overall effective register file size, resulting in an overall reduction of over 50%. In particular, both register file packing and register virtualization are orthogonal and combine in benefit, where both utilize indirection using a renaming table, amortizing this common overhead. This reduction in register file size can be leveraged for other optimizations, such as power gating unused registers to save static power [1], or enabling more kernel blocks/threads to be supported using the same register file to improve performance [56].

This paper makes the following contributions:

- We introduce the idea of register read coalescing, enabling the combination of multiple register reads into a single physical read. CORF implements coalescing with the aid of compiler-guided hints to identify commonly occurring register pairs.
- We propose CORF++, consisting of a re-organized register file to enable coalescing across different physical registers, and a compiler-guided allocation policy that optimizes allocation against this new register file. This new policy relies on compile-time graph coloring analysis, solving the bipartite edge frustration problem.
- We combine CORF++ and register file virtualization, observing that their benefits add up (CORF++ optimizes in space, while virtualization optimizes in time), but their overheads do not (both share a renaming table), resulting in the smallest known effective register file size among register compression proposals.

2 Background

In this section, we first overview the organization of modern GPU register files as well as its impact on performance and power. Next, we discuss the concept of register packing [16, 56], from which register coalescing opportunities arise.

GPU Register File: Modern GPUs consist of a number of Streaming Multiprocessors (SMs), each of which has its own register file, and a number of integer, floating point, and specialized computational cores. A GPU kernel, i.e. program, is decomposed into one or more Cooperative Thread Arrays (CTAs, also known as thread blocks) that are scheduled to the SMs. The threads within a block are grouped together into *warps*, or *wavefronts*, typically of size 32. The threads within a warp execute together in lockstep, following a Single Instruction Multiple Thread (SIMT) programming model. Each warp is assigned to a warp scheduler that issues instructions from its pool of ready warps to the operand collection unit (OC) and then to the GPU computational cores.

Each warp has its own set of dedicated architectural registers indexed by the warp index. There is a one-to-one mapping between architectural registers and physical registers [29]. To provide large bandwidth without the complexity of providing a large number of ports, the register file is constructed with multiple single-ported register banks that operate in parallel. A banked design allows multiple concurrent operations, provided that they target different banks.

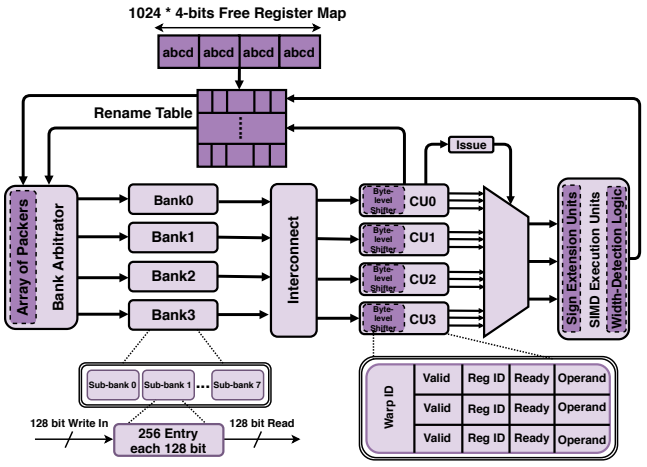


Figure 1. Baseline GPU register file design with proposed enhancements (in dark purple) for register packing [16, 56]. CU0-CU3 are operand collector units.

When multiple operations target registers in the same bank, a *bank conflict* occurs and the operations are serialized.

Figure 1 shows our baseline register file organization for the Fermi generation of Nvidia GPUs. It has a register file size of 128 KB per SM split across four banks. A bank is made up of 8 sub-banks that are 128 bits wide each. All 32 registers belonging to the 32 threads in the same warp are statically allocated to consecutive sub-banks (in a single bank) with the same entry index. Thus, a full register for all the threads within a warp can be striped using one entry of one bank, allowing it to be operated on in a single cycle. Each bank can store up to 256 warp-registers.

Impact of Register File on Performance and Power: When a warp instruction is scheduled by the warp scheduler, an operand collector (OC) unit is assigned to collect its operands for execution. An OC fetches the register operands from the register banks they reside in, bound by the two following constraints: (1) *OC port serialization*: Each OC has only one port and therefore it has to serialize reads when an instruction has multiple operands (instructions may need up to 3 source operands); and (2) *Register bank conflicts*: While operands from different banks may be concurrently read from different OCs, operands that access the same bank cause bank conflicts and cannot be issued together. The port constraints causing these conflicts are difficult to bypass by increasing the number of ports [9]: the cost of a port is extremely high when considering the width of a warp register. Register coalescing can help with both of these constraints: by coalescing operands, it allows multiple operands to be read by an OC in a single cycle, overcoming port serialization. Moreover, by reducing the overall number of register reads, the pressure on the register file is reduced, potentially reducing register bank conflicts. By reducing the overall number of reads to the RF, energy efficiency is improved. Moreover, improving performance leads to shorter run times, also improving energy efficiency.

Register Packing: Register coalescing opportunities arise when two registers needed by the same instruction are stored in the same physical register entry. This opportunity only exists when we allow multiple registers to be packed in the same physical register entry, a known architectural technique called *register packing* [16, 56]. In particular, register packing maps *narrow-width values* (values which do not need all 32 bits to be represented) of multiple architectural registers to a single physical register.

Since each architectural register read in prior register packing implementations requires a separate uncoalesced physical register read, a greedy *first-fit allocation policy* has been utilized to pack registers. This simple policy is sufficient to achieve the main goal of register packing, which is reducing the effective register file size; enabling unused registers to be power gated, or enabling the register file to be provisioned with a smaller number of physical registers. However, as we will show in the next section, this policy leads to very few register coalescing opportunities. Thus, a key to register coalescing is to pack related registers that are frequently read together, which is the goal of our compiler analysis.

3 The Virtues of Register Coalescing

In this section, we motivate register coalescing, and the need to design coalescing-aware register files to maximize the benefits of register coalescing. All experiments are collected with the GPGPU-Sim simulator [7], modeling a Fermi GPU[2]. We utilize benchmarks from Rodinia [15], Parboil [53], NVIDIA CUDA SDK [39], and Tango DNN Benchmark Suite [22]. More details of experimental setup are discussed in Section 7.

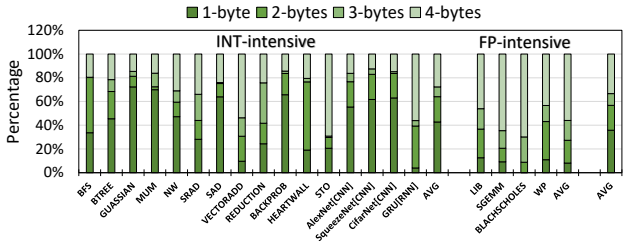


Figure 2. Width distribution of registers accessed from RF

Register operand characteristics: Figure 2 showcases the prominence of narrow-width values in GPU applications. We classify narrow-width values into four size classes: 1 byte, 2 bytes, 3 bytes, and 4 bytes (full-width). On average, 65% of all register operations contain narrow-width values, with over 33% of operations consuming no more than a single byte. This demonstrates that there exists a significant amount of register operands amenable to register coalescing. For floating point (FP)-intensive benchmarks (such as *sgemm* and *blackscholes*), the percentage of narrow-width values is

[2]Register coalescing opportunities are agnostic to hardware architecture.

less than that for integer-intensive benchmarks (such as *bfs* and *btree*). This is due to the IEEE 754 encoding of floating point values, which makes use of all 32 bits.

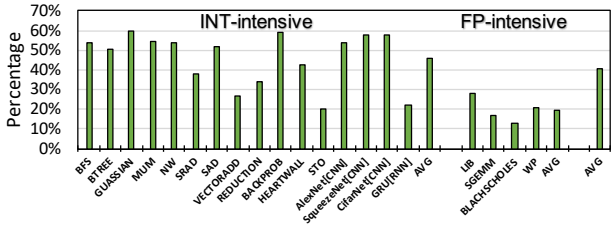


Figure 3. Unused RF bandwidth (also proportional to wasted dynamic energy).

Opportunity– Register file bandwidth: Figure 3 shows the unused register file bandwidth due to carrying the unneeded bits of narrow-width values. In addition to wasting bandwidth, these unneeded bits also cause wasted dynamic energy, as they are unnecessarily carried through to the operand collector. We observe more wasted bandwidth in integer applications, since narrow-width values are more common in them than in floating point applications.

While register packing is able to reduce the effective size of the register file, each register read still requires a separate physical register read. Therefore, this wasted bandwidth is not recovered with simple register packing. To this end, our proposed register coalescing aims to read multiple related registers used by the same instruction through a single register read operation in order to utilize the register file bandwidth more efficiently.

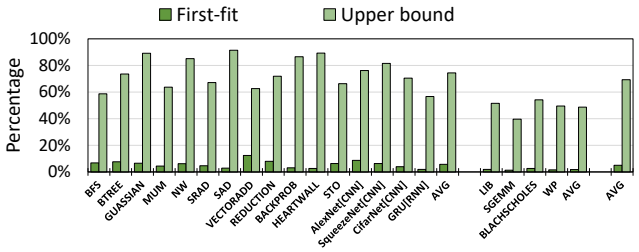


Figure 4. Instructions with coalesceable register reads; first fit is clearly weak in promoting coalescing.

Register coalescing opportunity: Figure 4 shows the prevalence of operand coalescing opportunities. We profile the register operand values at run-time and measure the fraction of all dynamic instructions which contains two register source operands that are both narrow and can fit together in a single register entry. We consider instructions that have two or more register source operands because they could benefit from coalescing. We find that around 40% of the instructions have two or more register source operands, but more importantly, because they read multiple registers, they are responsible for over 70% of the register file reads. On average, 69% of all dynamic instructions with two or more operands have the potential for operand coalescing, because their register operands can be packed, with up to 91% in some benchmarks like *Sad* and *Gaussian*. Clearly, we have more coalescing opportunities in integer intensive applications compared to floating point.

If we extend register packing to enable coalescing but keep the greedy first-fit register allocation policy, we can only leverage register coalescing opportunities in around 4% of instructions with two or more operands. This is a tiny fraction of the 69% of such instructions where a coalescing opportunity is potentially available! To improve coalescing opportunities, CORF incorporates a compiler-guided register allocation policy to identify pairs of registers commonly read from the same instruction and map them into the same physical register entry. In addition, we propose a coalescing-aware register file sub-bank organization and associated compiler-guided allocation policy (CORF++) which can coalesce register operands that are *not* stored in the same physical register entry, but in non-overlapping byte slices in the sub-bank.

4 CORF: Coalescing Operands in Register File

In this section, we present the design of CORF, which coalesces register reads to improve the RF performance. For two reads to be coalesceable, they have to be destined to registers that are packed in the same physical register entry. To improve the opportunity for coalescing, CORF utilizes compiler-assisted hints to pack related registers together. CORF is the first register file optimization technique that simultaneously improves performance and reduces power (both leakage and dynamic power). Coalescing enables higher performance by combining read operations, reducing port serialization of operand collector units and register file port conflicts. Coalescing reduces dynamic power, by decreasing the number of read operations to the register file, and lowers the overall GPU energy consumption because it leads to overall performance improvement that enable programs to finish faster. In Section 5, we will present CORF++, which further re-architects the register file organization to create more coalescing opportunities.

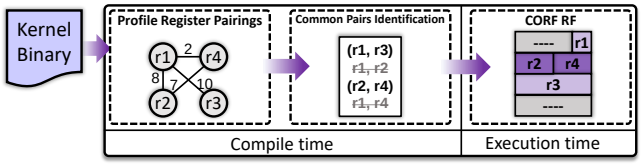


Figure 5. CORF overview. Compiler-generated register pairs guide register allocation to create coalescing opportunities

4.1 CORF Overview

CORF identifies *register pairs*—registers that are used as source operands in the same instruction—at compile time through static analysis or, alternatively, profiling. For example, in Figure 5, we have four registers (`r1`, `r2`, `r3`, `r4`), where

register r1 is read 8 times with r2, 10 times with r3, and 2 times with r4. In this example, we select (r1, r3) and (r2, r4) as target exclusive common pairs for coalescing. During run-time, if any of these common pairs happen to be compatible narrow-width values, they will be dynamically packed together. If any instruction requires both r2 and r4 as source operands, we can coalesce the operand access using a single read of the register file. However, in this example, during run-time (r1, r3) could not be packed since their combined size exceeds the size of a physical register entry. Since each register can only be coalesced with at most one other register, we lose opportunities to coalesce operands from instructions with different register pairings, such as (r1, r2), a limitation which we will target in Section 5.

4.2 Generating Compiler-assisted Hints

Identifying exclusive common pairs: The first step in identifying common pairs is to profile the frequency of register pairings in order to build a *Register Affinity Graph*, as shown in Figure 5. In order to determine the edge weights, we task the compiler to estimate the dynamic frequency of occurrence for each instruction in each kernel. This is, in general, a difficult problem at compile time, which we approximate as follows. For each instruction outside of a loop with two or more operands, we consider every pair of operands to occur once. Inside of loops, if the loop iteration count is statically resolvable, we use that count to increment the edge weight for register pairs that occur in the loop. If the iteration count is not a resolvable constant, we give a fixed weight to each register pair in instructions inside the loop. We use the same approach for nested loops. While these weights are not exact, they serve as a heuristic to assign relative importance to register pairs.

In order to identify exclusive common pairs, we must remove edges of the registers that have more than one edge. Considering only registers with more than one edge, we repeatedly remove the edge with the least weight until we end up with only exclusive pairs of registers. If there are any pair of registers that have all of their edges removed, we check if an edge can be restored between them.

Passing compiler-assisted hints to hardware: The set of exclusive register pairs that are identified by the compiler are annotated in the executable's preamble of a kernel and delivered to the hardware through a metadata instruction. The register pair information is maintained in a small associative structure. Specifically, we use a 64-bit metadata instruction (to be aligned with existing SASS binaries) in the beginning of each kernel in order to carry the compiler hints to the hardware. Consistent with the SASS instruction set that uses 10 bits as opcode for each instruction, we reserved 10 bits as opcode and the remaining bits for storing the common pairs of the registers. Since in Fermi architecture, each thread may have up to 63 registers, we need 6 bits as the register number. Each metadata instruction can carry up to four common pairs. Multiple instructions are used if more than 4 pairs need to be communicated. This design can also be adapted to support newer GPUs with more registers.

4.3 CORF Run-time Operation

We complete the description of CORF by explaining how registers are allocated to control the allocation of compiler identified pairs. We will also describe how coalescing opportunities are identified.

CORF register allocation policy: The register allocation policy for CORF attempts to pack the identified register pairs into the same physical register entry to increase coalescing opportunities. A register is allocated for the first time it appears as the destination of an instruction. Additionally, it could be reallocated when its size changes. When an allocation event occurs, we check the register pair information to see if the register belongs to a common pair. If it is, the allocator uses the common pair allocation logic. If the register does not belong to a common pair, it is allocated using the default allocation policy (assumed to be first-fit). We illustrate the common pair allocation using an example. Assume that r1 and r2 are identified as a common pair. When the first operand (say r1) arrives and is to be allocated, it is identified as a common pair register and mapped to any free full-width physical register. The rationale is to reserve any remaining slices of the physical register for a future allocation of the other register in the pair. When the buddy register (the register complementing the pair, which is r2 in this example) is allocated, we check to see if it fits the physical availability in the register allocated to r1. If it fits, it is allocated to the same physical register. Otherwise, it is mapped using the default policy.

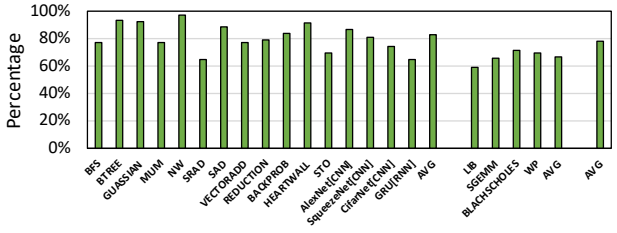


Figure 6. Percentage of successful combinations of compiler identified register pairs for CORF

In Figure 6, we show that identified common pairs fit together, and are successfully packed in the same register in most of the cases (an average of just under 80%). This is a high percentage despite the fact that we currently carry out no size estimation in the compiler analysis.

Identifying coalescing opportunities: Recall that packing registers in the same physical register is enabled by a *renaming table* (RT) that maps the architectural register to the physical register slice where it is stored. The RT is indexed by a tuple of the warp ID and an architectural register number. Each physical register is split into four 1-byte slices.

Thus, each RT entry stores the physical register where this value is stored, and a 4-bit vector called the *allocation mask*, which specifies the bytes in the physical register that the potentially narrow architectural register resides in. We use a *free register map* to keep track of free allocations of physical register slices when making allocation decisions. The free register map is a bit-vector where each bit represents a byte of a physical register (i.e., 4 bits per physical register).

To identify coalescing opportunities as a new instruction is sent to an operand collector unit, we first look it up in the renaming table to determine the physical registers where the operand registers are stored. If the physical registers for two operands match, the reads to these operands are coalesced into a single read to the register file. When the physical register contents are received, the unpacker demultiplexes the two registers and sign-extends them to recover two full-length registers.

Incorporating register virtualization [21]: CORF's implementation seamlessly supports register file virtualization to further reduce the size of the register file. Specifically, we observed that register file virtualization, which releases registers when they are no longer live, can also further reduce the register file size. At the same time, register file virtualization can be directly supported within CORF since it also relies on a renaming table to allocate registers, requiring almost no additional overhead.

5 CORF++: Re-architected RF

CORF coalescing opportunities are limited to registers stored within the *same physical register entry*. If a register is commonly accessed with two or more other registers, coalescing is possible with only one of them. To relax this limitation, CORF++ re-organizes the register file to enable more operand coalescing opportunities.

Specifically, CORF++ (Figure 7) re-architects the register file to enable coalescing of registers within the *same physical register bank*, provided they reside in *non-overlapping sub-banks*. Recall that each bank consists of eight sub-banks of 16 bytes wide. Since we are no longer restricted to coalescing exclusive pairs of registers packed into the same physical register entry, the compiler's task of guiding register allocation to promote coalescing becomes substantially different. In this section, we overview CORF++. We first present the compiler support to optimize coalescing opportunities in CORF++, then describe the implementation of the coalescing aware register file, and finally discuss its operation during run-time.

5.1 Compiler-assisted Register Allocation

CORF++ allows coalescing registers in non-overlapping sub-banks, even if the values reside in two different physical register entries. The main challenge of efficient register allocation in CORF++ is in assigning commonly read register

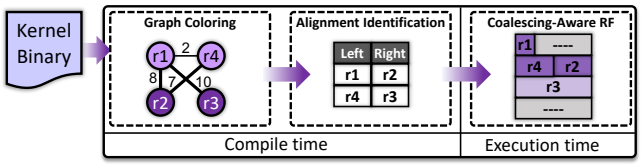


Figure 7. CORF++ overview. At compile time, we identify which registers should be left-aligning, or right-aligning through graph coloring algorithm, so that we can maximize coalescing opportunities. This information will then guide register allocation in our coalescing-aware register file.

pairs in different sub-banks. We simplify the allocation to a selection of left-aligning and right-aligning assignments; provided that two registers are in separate alignments, they have a chance of being coalesced (subject to their combined size being smaller or equal to 4 bytes).

Similar to the compiler analysis for CORF, we start by constructing the Register Affinity Graph where edges between registers indicate the expected frequency of reading the two registers together in the same instruction. An optimal assignment maximizes the weight of the edges between registers assigned to alternate alignments. This problem maps to a graph coloring problem variation (where each alignment is a color). We are attempting to remove the minimum edge weight (thus, forsaking the least coalescing opportunities) to enable the graph to be colorable by two colors (left or right). This variation of graph coloring is called the bipartite edge frustration problem, and is NP-hard even with two colors [61].

To derive an efficient heuristic for register mapping, we first observe that any graph with no odd cycles (cycles made up of an odd number of edges) is 2-colorable. Thus, to solve the problem, we should remove the minimum set of edges, considering weight, that will break all odd cycles (to identify odd cycles, we used a modified version of the algorithm in [12]). Since the optimal solution is NP-hard, we develop the following heuristic, as illustrated in Figure 8. In the initial graph state (left-most graph), we have four odd cycles: (`r1,r2,r3`), (`r3,r4,r6`), (`r2,r3,r4,r6,r5`), and (`r1,r3,r6,r5,r2`). We assign each edge a weight corresponding to its original weight, divided by the number of odd cycles that removing it would break. We then remove the edge with the minimum weight (among the edges that are part of odd cycles), and update the weights. We repeat this process until all odd cycles are eliminated, enabling us to trivially 2-color the graph.

Similar to CORF, the register allocation information is passed through metadata instructions. We use a metadata instruction to encode the register assignments to either left-aligning, right-aligning, or don't-care. This encoded data is expanded to store 2 bits per register to indicate alignment. This data is stored using a single bit-vector for each kernel, resulting in a storage overhead of 128 bits per kernel. Other

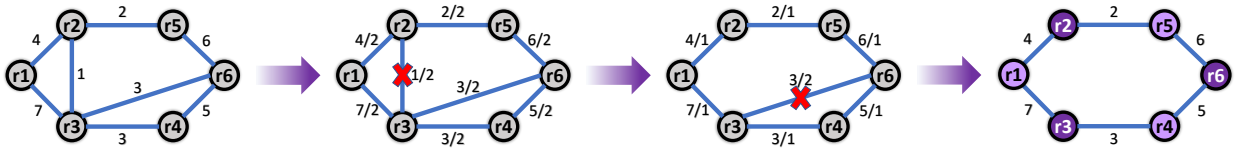


Figure 8. CORF++ register assignment heuristic example

designs that reduce or completely remove this overhead are possible, for example, having the compiler preset the register alignments (e.g. all even registers right aligned).

5.2 Coalescing-aware RF Organization

Mapping registers to banks: In the baseline register file, registers belonging to the same warp are interleaved across the register banks with the goal of minimizing bank conflicts across warps (Figure 9, left side). Since coalescing occurs only within a single instruction of a warp, CORF++ maps all registers belonging to the same warp to a single register bank in order to maximize coalescing opportunities (Figure 9, right side). This new mapping ensures that all accesses to registers within the same warp are in the same bank and therefore potentially coalesceable.

Counter-intuitively, our goal is to create more bank conflicts within warps, which gives us more opportunities to convert bank conflicts into beneficial coalescing opportunities. Note that since the operand collector unit can read no more than one register in each cycle, there is no lost opportunity in terms of reading registers from different banks for the same instruction. With respect to conflicts across warps, on average, the new mapping does not increase conflicts, since the probability of two registers from two different warps being in the same bank remains $\frac{1}{n}$, where n is the number of banks. However, with the new mapping, two warps either always conflict (because they are mapped to the same bank) or they never do (mapped to different banks) and there is a possibility for pathologies arising, for example, from two active warps being mapped to the same bank. However, we did not observe any such behavior in our experiments.

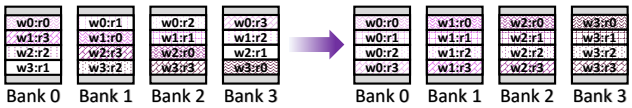


Figure 9. Modified register to bank mapping where all registers belonging to a warp maps to the same bank.

Sub-bank organization: CORF++ allows multiple read operations to registers that reside in non-overlapping sub-banks to be coalesced. To support this functionality, we change the mapping of the registers to sub-banks. For clarity, we denote the bytes of a 32-bit register values as $B_3B_2B_1B_0$.

In Figure 10 A we show how registers are organized across the 8 sub-banks in current GPUs. A register is stored across all 8 sub-banks, where each sub-bank is 128 bits wide. Each sub-bank stores a 32-bit register value for 4 threads. For example, sub-bank 0 stores the register values for threads 0 - 3 in sequential order, where the first 4 contiguous bytes are from thread 0, the next 4 bytes are from thread 1, and so on.

Now let us assume that `r1` and `r4` are 1-byte narrow values, and `r2` and `r3` are 3-byte narrow values. Figure 10 B shows how these four architectural registers are stored after they are packed into two physical registers. For example, in physical register $P0$, `r1` and `r3` are packed together. In this example, since `r3` is 3-bytes, `r3` will only utilize the 3 least significant bytes (B_{2-0}). This mapping leaves the most significant byte (B_3) available, which is packed with `r1`. `r2` and `r4` are also packed similarly. In this scenario, we can only coalesce reads if they require `r1` and `r3`, or `r2` and `r4`, as these pairs reside in the same physical register entry. Here we lose coalescing opportunities for other compatible pairs, such as `r1` and `r2`, or `r3` and `r4` since parts of every register are spread across all sub-banks.

To address this limitation, we present a re-organized sub-bank mapping, as shown in Figure 10 C. Instead of storing registers in sequential ordering of the entire 32-bit register value, we will instead interleave the storage of register values across the sub-banks. In this scenario, we first store the most significant bytes (B_3) of threads 0 - 31 consecutively, then store the next significant bytes (B_2) of threads 0 - 31, etc. In this organization, B_3 is stored in sub-banks 0 and 1, B_2 is stored in sub-banks 2 and 3, and so on.

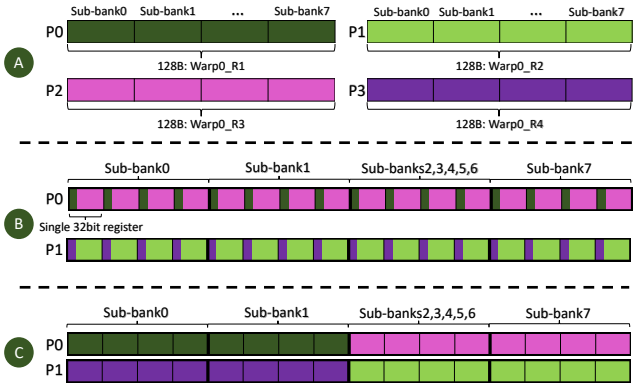


Figure 10. Baseline register sub-bank organization shown in A. Sub-bank organization when packing R1 w/ R3, and R2 w/ R4 B. Coalescing-aware sub-bank organization C enables coalescing across different physical registers with non-overlapping sub-banks.

When storing packed values in CORF++, we store the narrow registers as either *left-aligning*, or *right-aligning*. In the case of r1 and r3, r1 is stored into *P*0 as left-aligning, and r3 is stored as right-aligning. In this new sub-bank organization, we are able to coalesce r1 and r3, and r2 and r4. Note that if each sub-bank can address different physical register addresses, then it would also be possible to coalesce registers in non-overlapping sub-banks. For example, r1 and r2, as well as r3 and r4 would be coalesceable.

Dual-addressable banks: To support coalescing across different physical register entries, we introduce dual-addressable banks (Figure 11). We add additional MUXes to pick between *Address*1 and *Address*2, which represent a left-aligning and a right-aligning register being coalesced. If we wish to coalesce r1 and r2, then *P*1 would be sent to *Address*1, and *P*0 to *Address*2. By default, the MUXes select *Address*1, and utilize the 4-bit allocation mask from *Address*2's entry in the renaming table as the selector. In this scenario, we use r1's allocation mask, which would be 1000.

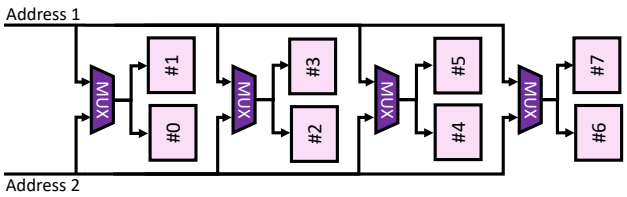


Figure 11. Dual address register file.

5.3 CORF++ Run-time Operation

Next, we explain the run-time operation of CORF++ through an illustrative example to demonstrate register allocation and coalescing.

CORF++ register allocation: When an allocation event occurs (e.g., writing into r2 in Figure 12 Ⓑ), we check the register alignment to see if it is a right-aligned or left-aligned register. For don't-care registers, we default to the first-fit allocation.

Identifying coalescing opportunities: Similar to CORF, to identify coalescing opportunities as a new instruction is sent to an operand collector unit, we look up the allocation mask in the renaming table for the source operands. Any two source operands could be coalesced if the AND of their allocation masks becomes 0000.

Figure 12 shows an illustrative example of CORF++ with three physical registers. Ⓐ shows a piece of SASS code. The value loaded in r1 in Ⓑ is detected by a *width detection unit* as a narrow-width value that needs 2 bytes, and since r1 is an unallocated don't-care register, we map it to the first available spot (using first-fit policy). The next instruction writes into r2 which is right-aligned, so we map it to the first available *right* part of a physical register. In Ⓒ, the instruction writes into r4 and is allocated to the first available *right* part of a physical register. Ⓓ shows a local load into r3, so we map it to the first available *left* spot (which is *P0*). In Ⓔ, we first coalesce the read operation for r2 and r3 and then write into r5, so the allocator maps it to the first available *left* spot. Finally, in Ⓕ, CORF++ coalesces the read operations for r4 and r5 and later r3 and r4. In this example, we were able to coalesce all available opportunities. In contrast, CORF is not able to coalesce read operations for r3 and r4 because we can only pick *exclusive common pairs*.

6 Additional Implementation Details

CORF assumes as a starting point a register file that implements register packing RF [16, 56] and extends it in three important ways: (1) It supports operand coalescing: the ability to identify opportunities for reading registers that are packed in the same physical register (CORF) or in mutually exclusive sub-banks (CORF++), and the support to read them together and unpack them; (2) It receives compiler hints to guide register allocation decisions and uses them to guide allocation to promote coalescing; and (3) It also supports register virtualization [21], allowing it to free registers when they cease to be live. Additionally, CORF++ rearchitects the register file to enable coalescing reads from mutually exclusive sub-banks as we described in the previous section. In this section, we describe additional important components of CORF and CORF++.

Renaming Table (RT): The renaming table is a table indexed by a tuple of the warp ID and an architectural register number. Each entry stores the physical register where this value is stored, and a 4-bit allocation mask. The table consists of ($max\_num\_of\_warps\_per\_SM \times max\_regs\_per\_thread$) entry, which is $48 \times 63 = 3024$ in our reference register file. Each entry has a width of 14 bits (10 bits to represent the physical register number, and the 4-bit allocation mask).

The renaming table needs to be accessed on register reads to resolve the mapping to the physical register. The number of ports needed must at least match the number of read ports on the register file to keep port conflicts from becoming a bottleneck. The renaming table can be implemented as a general multi-ported table. However, to reduce complexity, we implement it as a dual-ported sub-banked structure. We use two ports to allow fast lookup of potentially coalesceable registers. We use a design with a separate bank for each register file bank in the corresponding register file.

Allocation Unit: A small structure that guides the allocation policy using information provided by the compiler. We designed and synthesized this structure in detail for CORF++. It holds an allocation vector that carries the alignment for each register (left, right or don't-care). We store 128 bits per each kernel, for a maximum storage size of 128 bytes per SM (please note that we may have up to 8 concurrent kernels running on each SM). The allocation vector is consulted during allocation in conjunction with a free map that keeps track of the available physical registers (and register slices). The

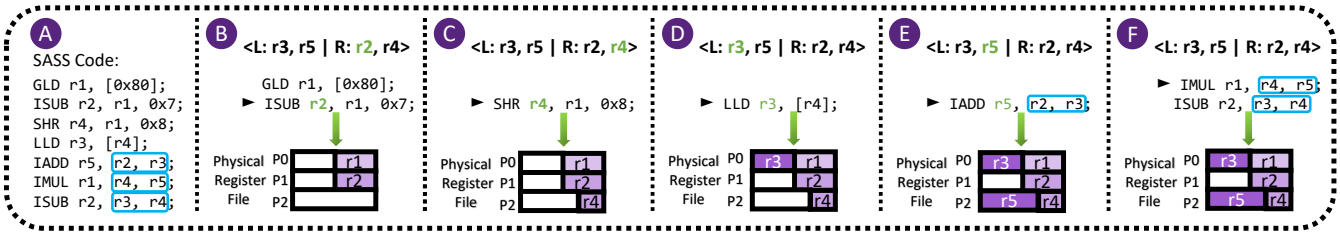


Figure 12. Illustrative Example of CORF++ register allocation (Ⓑ–Ⓓ) and read coalescing (Ⓔ, Ⓕ).

allocator logic uses the alignment preference as it consults the free map to identify a target register for allocation. Note that the renaming logic, free map, and the allocation logic are present in baseline register packing [16, 56]; our allocation unit adds the compiler hints and changes the allocation logic to use them.

Impact on pipeline: Although the RT access latency is low (0.38ns according to CACTI [52], which is well below the cycle time of modern GPUs), we want to avoid combining the RT lookup, coalescing logic, and the register file read in the same cycle. We note that once the scoreboard marks an instruction to be ready to issue, we need at least one cycle to find a free operand collector and move the instruction to it. Thus, we use this cycle to initiate access to the renaming table to avoid trying to fit the renaming table access and the register file access in the same cycle. The RT is dual-ported and sub-banked; however, in the event of a port conflict, the arbitrator (which resolves conflicts for the register file) is extended to delay the register read while the renaming table read is resolved. We extended the pipeline in the simulator to model these effects.

Control divergence: When control divergence occurs, only a subset of SIMT lanes of a warp are active. CORF operation continues unchanged under divergence but considering all registers (whether belonging to active or inactive threads) for all operations (importantly for width determination).

Size changes: If a packed narrow-value register size increases during runtime, we reassign it to another physical register entry using the same process as the initial assignment. The original mapping is then cleared. Size change events which require reallocation are rare (less than 0.3% of writes), which makes these extra accesses to the RT have negligible effects. In case of a size decrease, we keep the old mapping and adjust only the size in the renaming table.

Packers and unpackers: Packers and unpackers are placed as shown in Figure 1 so that packed values only exist in the register file and operand collection pipeline stage. Registers are packed as they are written to the register file by first aligning them into the slice they will be written to, and writing only that slice of the physical register. Conversely, when registers are read, they are unpacked by shifting down (if necessary) and sign-extending such that the registers are recovered to full width. Our unpackers are designed to be able to unpack two values in the case of coalesced reads. The number of packers required matches the pipeline width for writing (in our case, two packers). To unpack coalesced registers, we have two unpackers working in parallel in each operand collector, for a total of 8 unpackers per SM.

Width detection units: The register width detection units are embedded into the final stage of SIMD execution units in order to detect the width of produced outputs. This is a combinational circuit: it ORs the 7 least significant bits for each of the three most significant bytes for every register in addition to the most significant bit of the byte before it (to ensure that narrow positive numbers always start with a 0 in the MSB). For example, for byte 1 which spans bits 8 to 15, we OR together bits 7 to 14 to identify whether the byte is 0 or not. This produces a 3-bit output for each register. Moreover, another 3 bits are obtained by NAND-ing together the same bits of each byte to track the width of negative numbers. Again, this ensures that any shortened negative number has 1 in the MSB. We use the most significant bit of the register to multiplex out either the OR outputs (for positive values) or the NAND outputs (for negative values). A second stage ORs the 3 bits output of the MUX per register across all 32 registers in the warp producing a single 3-bit output to capture the maximum width. This 3-bit sequence is used to determine the overall size of the register.

7 Performance/Power Evaluation

We have implemented CORF and CORF++ in GPGPU-Sim v3.2.1 [7], based on an Nvidia Fermi-like GPU configuration with 15 SMs. Each SM has a 128 KB register file organized into four banks, and each bank consists of eight sub-banks, as detailed in Figure 1. We enabled PTXPlus for all of our evaluations. Since GPGPU-Sim provides a detailed PTX code parser, we modified the parser to carry out our compiler optimizations. Each SM also has two warp schedulers configured to use a two-level warp scheduler.

In all experiments, we use 20 benchmarks selected from Rodinia [15], Parboil [53], NVIDIA CUDA SDK [39], and Tango [22] benchmark suites. The benchmarks cover a range of behaviors and operand mixes (integer/floating point).

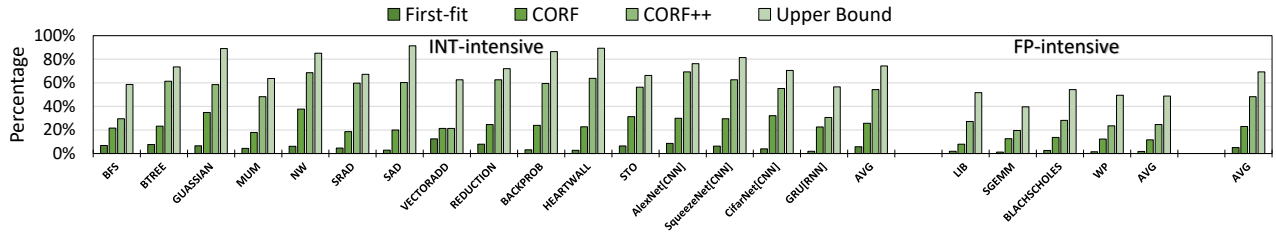


Figure 13. Coalesced instructions: CORF and CORF++ significantly increases the amount of coalescing opportunities.

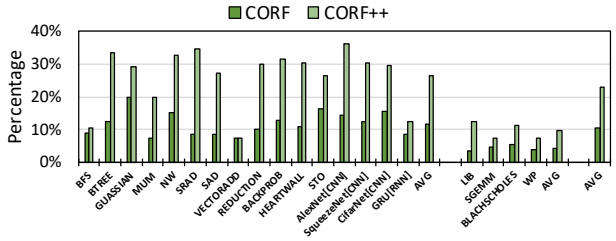


Figure 14. Reduction in number of accesses to register file.

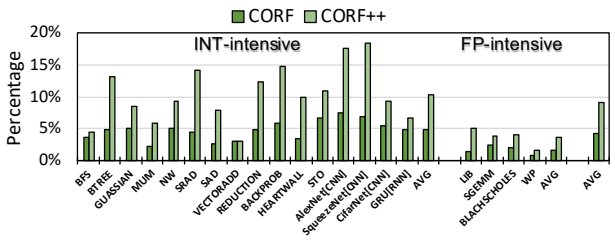


Figure 15. IPC Improvement.

Coalescing success: Figure 14 shows the reduction in register file accesses due to operand coalescing in CORF and CORF++. CORF reduces the overall number of register file accesses, by 12% for integer applications, 4.5% for floating point applications, and 10% of all applications. This reduction percentage is computed against all accesses (including writes, and instructions with a single register operand, which cannot be coalesced). CORF++ is able to reduce even more accesses (by 2.3x) because of increased coalescing opportunities. Specifically, CORF++ reduces register access of integer applications by 27%, floating point applications by 9.9%, and 23% overall. Figure 13 shows the impact of compiler optimizations on the success of coalescing. While first-fit allocation policy results in coalescing only 4% of the instructions with multiple register operands, CORF and CORF++ are able to coalesce 23% and 48%, respectively.

Performance: As a result of the reduced register accesses, performance is improved. Figure 15 shows the performance impact of CORF and CORF++. Notably, we observe IPC improvement across all benchmarks. On average, CORF improves IPC by 4.9% for integer benchmarks and 1.7% for floating point benchmarks (harmonic mean across all applications is 4%). For fairness, the IPC computation does not count metadata instructions since they do not further the computation (but we include their cost). CORF++ is able to improve IPC for integer benchmarks by 10.5%, floating point ones by 3.6%, resulting in a harmonic mean of 9%.

Register file size: A secondary contribution of CORF is that we combine register packing and register virtualization to reduce the overall register file size beyond either of these techniques alone. Virtualization is essentially obtained for free since it primarily relies on a renaming table such as the one we already use. Figure 16 shows the reduction in the number of allocated physical registers using register packing, register file virtualization (RF-Virtualization) [21], and when combined together. We tracked the number of allocated physical registers (each potentially packing several architectural registers) as a fraction of the total number of architectural registers averaged over the benchmarks' execution. Register packing reduced physical-register allocation by 34%, register file virtualization alone reduced it by 35%, while both together reduced it by 54%. When combined, packing compresses spatially, and RF-Virtualization temporally, leading to synergistic improvements [10, 11]. This is the highest compression ratio achieved by techniques that attempt to compress the register file size [21, 27, 56]. The reduction in effective register file size can be exploited either: (1) by gating unused registers to save power; (2) by reducing the register file size while maintaining performance; or (3) by enabling more threads to be active to improve performance. We demonstrate the advantage using the first option.

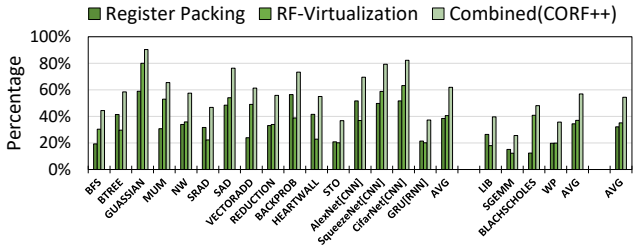
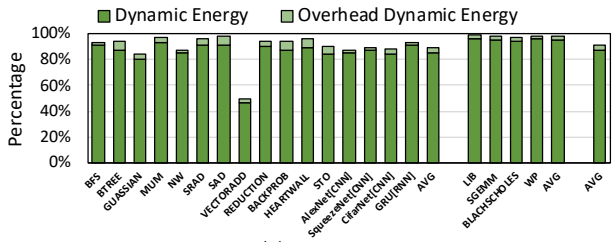
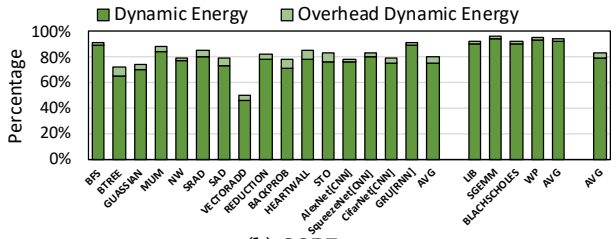


Figure 16. Reduction in allocated physical registers.

RF energy: Figures 17 and 18 show the dynamic energy and leakage energy impact of our techniques. The small segments on top of each bar represent the overheads of the structures added by CORF/CORF++. Dynamic energy savings in Figure 17 are due to the reduced number of accesses



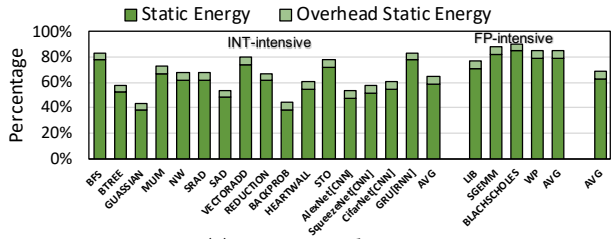
(a) CORF



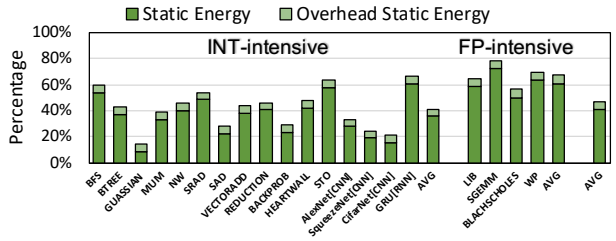
(b) CORF++

Figure 17. Normalized RF dynamic energy

to the register file because of operand coalescing. We observed 8.5% and 17% reduction to the overall dynamic energy in CORF and CORF++, respectively, after considering the 3% increase in overheads. The source of dynamic energy overheads include the packers and unpackers, width detection logic, and the accesses to the renaming table.



(a) Register packing.



(b) Combined with virtualization (CORF++).

Figure 18. Normalized RF leakage energy

Figure 18 shows the leakage energy for register packing and also the combined register packing and virtualization (CORF++), assuming that we power gate unused registers. Leakage energy is reduced by 33% in register packing (Figure 18a), and 52% for the combined with virtualization (CORF++, Figure 18b), after accounting for the overheads. On average, the leakage overhead, due to the additional structures (e.g. renaming table, free-register map), is 5.4%, which is easily out-weighed by the leakage energy savings.

Technique	IPC	Register Reads	RF Dyn. Energy	RF Size
Register Packing	1	1	1	0.65
Register Packing + Virtualization	1	1	1	**0.43**
CORF	**1.04**	**0.9**	**0.92**	**0.43**
CORF++	**1.09**	**0.77**	**0.83**	**0.43**

Table 1. Summary of CORF, CORF++, and register packing (and register virtualization). All values normalized to the baseline GPU register file.

We summarize the advantages of CORF/CORF++ compared to register files without coalescing in Table 1. Note that Wang et al. [56] evaluate the performance of register packing when they exploit the smaller effective register file to allow more threads to run concurrently per SM. This IPC improvement technique is orthogonal to coalescing and can be combined, therefore we do not include it for comparison.

8 Hardware/Software Overheads

Hardware overheads: The largest additional structure in CORF is the renaming table, which is also needed for simple register packing [16, 56]. Each RT entry consists of 14 bits that encodes the physical register and slice to which an architectural register is being mapped. Since our baseline architecture supports up to 48 warps per an SM, and 63 registers per warp, for a total of just over 3000 potential warp architectural registers. Each register has an entry in the table. Therefore, RT total size is $5.16KB$ which is 4% of total $128KB$ register file per each SM. The free register map size is $1024 \times 4 - bits$ or $512bytes$. Supported by the RT, register packing and virtualization reduce the effective register file size to less than half of its original size: *the benefits of shrinking the register file easily offset the overhead, before we even consider coalescing*. We calculate the renaming table and register file power consumption using CACTI v5.3 [52] and report them in Table 2.

The overhead of logic, such as the allocation policy logic, coalescing logic, packers, unpackers, and width detection units, was estimated by synthesizing its Verilog HDL description using Synopsys Design Compiler and the NCSU PDK 45nm library [36]. The static and dynamic energy of these logics are also included in our power results. All together, these logic accounts for 57mW of dynamic power, 0.2mW static power, and 0.05mm^2 (or 0.11%) of total on-chip area.

Parameter	Renaming table	Register bank	Percentage
Size	5KB	128KB	3.9%
# Banks	4	4	-
Vdd	0.96V	0.96V	-
Access energy	1.83pJ	149.76pJ	1.2%
Leakage power	5.56mW	89.6mW	6.2%

Table 2. Renaming table overheads in 40nm technology

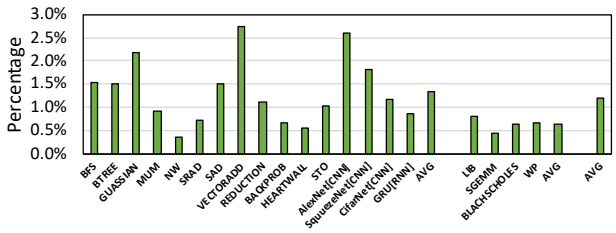


Figure 19. Static code size increase.

Software overheads: Figure 19 shows the static code increase due to the addition of extra instructions to guide CORF. Overall, CORF only increases the code size by 1.3%. Passing information in CORF++ can be simplified, for example, by having the compiler choose odd register numbers for the left operands, and even numbers for the right operands without explicit metadata instructions. When considering dynamic instruction count, this overhead will be significantly lower.

9 Related Work

Energy efficiency of GPU has been an area of increasing importance [1–5, 14, 23, 30–34, 46, 48–50, 54, 57–59]. These prior works have explored improving the performance or energy efficiency of GPU register files in a number of ways. In this section, we will highlight works related to GPU register files.

Warped Register File [1] introduces a tri-modal register file structure that enables drowsy mode. Pilot Register File [2] proposed an energy-efficient RF design using FinFETs. Register File Caching [17, 18] proposed to add a small register file cache to reduce overall RF dynamic power by storing frequently accessed registers in an energy-efficient cache. However, these techniques solely aim to reduce power, with the goal of achieving a negligible performance penalty.

Several works aim to improve the performance of register files. RegMutex [25] improved performance by sharing a subset of physical registers between warps during the GPU kernel execution. FineReg [42] achieved a higher number of concurrent CTAs by partitioning the register file into two regions, one for active CTAs and another for pending CTAs. Register file slicing [20] proposed to split the data path into two 16-bit slices, which enables the register to save power by power gating a slice if storing narrow-values, *or* to improve performance by fetching two 16-bit values. RF slicing fundamentally trades-off between a power-efficient mode, or a performance-enhancing mode.

Another commonly used energy efficiency technique is value compression [6, 27, 35, 44, 45, 51, 55, 60, 62]. Register File Compression [27], utilize base-delta-immediate (BDI) compression to compress data within an entry and power-gate sub-banks. While Register Packing [16, 56] compress narrow values to use less physical register entries, and power gates unallocated entries.

Wang et al. [56] were the first to propose register packing for GPUs. Specifically, they greedily pack narrow-value registers together to reduce register file space. They do not coalesce register reads – each register read still requires a separate physical register file read operation. Register file virtualization [21] reduces the number of allocated physical registers required (and power gate unallocated entries), through register liveness analysis. While achieving power savings, these techniques do not improve performance. In our work, by combining packing and virtualization and also harnessing coalescing opportunities, we achieve higher compression ratios, power savings, *and* performance improvements.

RegLess [26] replaces the register file with a smaller staging unit with the help of compiler annotations, leveraging the short-lived and long-lived behaviors of the register. RegLess achieves lower power and smaller register storage size while maintaining performance. The Latency-Tolerant Register File (LTRF) [47] similarly uses compiler-analysis to identify registers to move into a register cache, which enables tolerance of large register files. However, this higher performance comes at the cost of a larger, more power-hungry register file.

10 Conclusion

In this paper, we introduce the concept of register coalescing. We proposed CORF, a coalescing-aware register file design for GPUs that simultaneously reduces the leakage and dynamic access power, while improving the overall performance of the GPU. CORF achieves these properties by enabling the reads to multiple operands that are packed together to be coalesced, reducing the number of reads to the RF, and improving dynamic energy and performance. CORF combines compiler-assisted register allocation with a re-organized register file (CORF++) in order to maximize operand coalescing opportunities. Specifically, the new register file organization allows operands to be coalesced even if they reside in different physical registers, provided they reside in non-overlapping sub-banks. In addition, we show that our technique can be seamlessly integrated with register file virtualization to provide even more benefits. Overall, we save 17% of the dynamic energy of the RF, and 52% of the leakage energy; reducing the number of reads by 23% and improving IPC by 9%.

Acknowledgments

We would like to thank Marek Chrobak for identifying the theoretical structure and potential heuristics for the CORF++ register assignment problem. This work is partially supported by the National Science Foundation under Grants CNS-1422401, CNS-1619450, CNS-1619322, and CCF-1815643, as well as Air Force Office of Scientific Research (AFOSR) under Award No. FA9550-15-1-0384.

References

[1] Mohammad Abdel-Majeed and Murali Annavaram. 2013. Warped register file: A power efficient register file for GPGPUs. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*. IEEE, 412–423.

[2] M. Abdel-Majeed, A. Shafaei, H. Jeon, M. Pedram, and M. Annavaram. 2017. Pilot Register File: Energy Efficient Partitioned Register File for GPUs. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[3] Mohammad Abdel-Majeed, Daniel Wong, and Murali Annavaram. 2013. Warped Gates: Gating Aware Scheduling and Power Gating for GPGPUs. In *Microarchitecture (MICRO), 2013 46th Annual IEEE/ACM International Symposium on*.

[4] Mohammad Abdel-Majeed, Daniel Wong, Justin Kuang, and Murali Annavaram. 2016. Origami: Folding Warps for Energy Efficient GPUs. In *Proceedings of the 2016 International Conference on Supercomputing (ICS '16)*.

[5] AmirAli Abdolrashidi, Devashree Tripathy, Mehmet Esat Belviranli, Laxmi N. Bhuyan, and Daniel Wong. 2017. WIREFRAME: Supporting Data-dependent Parallelism through Dependency Graph Execution in GPUs . In *MICRO '17: Proceedings of the 2017 50th Annual IEEE/ACM International Symposium on Microarchitecture*.

[6] Alaa R. Alameldeen and David A. Wood. 2004. Adaptive Cache Compression for High-Performance Processors. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA '04)*.

[7] Ali Bakhoda, George L Yuan, Wilson WL Fung, Henry Wong, and Tor M Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*. IEEE, 163–174.

[8] Mohammad Bakhshalipour, Aydin Faraji, Seyed Armin Vakil Ghahani, Farid Samandi, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. 2019. Reducing Writebacks Through In-Cache Displacement. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 24, 2 (2019), 16.

[9] Mohammad Bakhshalipour, Pejman Lotfi-Kamran, Abbas Mazloumi, Farid Samandi, Mahmood Naderan, Mehdi Modarressi, and Hamid Sarbazi-Azad. 2018. Fast Data Delivery for Many-Core Processors. *IEEE Transactions on Computers (TC)* 67, 10 (2018), 1416–1429.

[10] Mohammad Bakhshalipour, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. 2018. Domino Temporal Data Prefetcher. In *International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 131–142.

[11] Mohammad Bakhshalipour, Mehran Shakerinava, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. 2019. Bingo Spatial Data Prefetcher. In *International Symposium on High-Performance Computer Architecture (HPCA)*.

[12] Etienne Birmelé, Rui Ferreira, Roberto Grossi, Andrea Marino, Nadia Pisanti, Romeo Rizzi, and Gustavo Sacomoto. 2013. Optimal listing of cycles and st-paths in undirected graphs. In *Proceedings of the twenty-fourth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 1884–1896.

[13] Preston Briggs. 1992. *Register allocation via graph coloring*. Technical Report.

[14] N. Chatterjee, M. O'Connor, D. Lee, D. R. Johnson, S. W. Keckler, M. Rhu, and W. J. Dally. 2017. Architecting an Energy-Efficient DRAM System for GPUs. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[15] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. Ieee, 44–54.

[16] Oguz Ergin, Deniz Balkan, Kanad Ghose, and Dmitry Ponomarev. 2004. Register packing: Exploiting narrow-width operands for reducing register file pressure. In *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 304–315.

[17] Mark Gebhart, Daniel R Johnson, David Tarjan, Stephen W Keckler, William J Dally, Erik Lindholm, and Kevin Skadron. 2011. Energy-efficient mechanisms for managing thread context in throughput processors. In *ACM SIGARCH Computer Architecture News*, Vol. 39. ACM, 235–246.

[18] Mark Gebhart, Stephen W Keckler, and William J Dally. 2011. A compile-time managed multi-level register file hierarchy. In *Proceedings of the 44th annual IEEE/ACM international symposium on microarchitecture*. ACM, 465–476.

[19] Lal George and Andrew W. Appel. 1996. Iterated Register Coalescing. *ACM Trans. Program. Lang. Syst.* 18, 3 (May 1996), 300–324.

[20] Syed Zohaib Gilani, Nam Sung Kim, and Michael J Schulte. 2013. Power-efficient computing for compute-intensive GPGPU applications. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*. IEEE, 330–341.

[21] Hyeran Jeon, Gokul Subramanian Ravi, Nam Sung Kim, and Murali Annavaram. 2015. GPU register file virtualization. In *Proceedings of the 48th International Symposium on Microarchitecture*. ACM, 420–432.

[22] Aajna Karki, Chethan Palangotu Keshava, Spoorthi Mysore Shivakumar, Joshua Skow, Goutam Madhukeshwar Hegde, and Hyeran Jeon. 2019. Tango: A Deep Neural Network Benchmark Suite for Various Accelerators. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE Press.

[23] Onur Kayiran, Adwait Jog, Ashutosh Pattnaik, Rachata Ausavarungnirun, Xulong Tang, Mahmut T. Kandemir, Gabriel H. Loh, Onur Mutlu, and Chita R. Das. 2016. μC-States: Fine-grained GPU Datapath Power Management. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation (PACT '16)*.

[24] Farzad Khorasani, Hodjat Asghari Esfeden, Nael Abu-Ghazaleh, and Vivek Sarkar. 2018. In-Register Parameter Caching for Dynamic Neural Nets with Virtual Persistent Processor Specialization. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 377–389.

[25] Farzad Khorasani, Hodjat Asghari Esfeden, Amin Farmahini-Farahani, Nuwan Jayasena, and Vivek Sarkar. 2018. Regmutex: Inter-warp gpu register time-sharing. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE Press, 816–828.

[26] John Kloosterman, Jonathan Beaumont, D Anoushe Jamshidi, Jonathan Bailey, Trevor Mudge, and Scott Mahlke. 2017. Regless: just-in-time operand staging for GPUs. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 151–164.

[27] Sangpil Lee, Keunsoo Kim, Gunjae Koo, Hyeran Jeon, Won Woo Ro, and Murali Annavaram. 2015. Warped-compression: Enabling power efficient gpus through register compression. In *ACM SIGARCH Computer Architecture News*, Vol. 43. ACM, 502–514.

[28] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M Aamodt, and Vijay Janapa Reddi. 2013. GPUWattch: enabling energy optimizations in GPGPUs. In *ACM SIGARCH Computer Architecture News*, Vol. 41. ACM, 487–498.

[29] John Erik Lindholm, Ming Y Siu, Simon S Moy, Samuel Liu, and John R Nickolls. 2008. Simulating multiported memories using lower port count memories. US Patent 7,339,592.

[30] Z. Liu, S. Gilani, M. Annavaram, and N. S. Kim. 2017. G-Scalar: Cost-Effective Generalized Scalar Execution Architecture for Power-Efficient GPUs. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[31] Zhenhong Liu, Daniel Wong, and Nam Sung Kim. 2018. Load-Triggered Warp Approximation on GPU. In *Proceedings of the 2018 International Symposium on Low Power Electronics and Design (ISLPED '18)*.

[32] A. Majumdar, L. Piga, I. Paul, J. L. Greathouse, W. Huang, and D. H. Albonesi. 2017. Dynamic GPGPU Power Management Using Adaptive Model Predictive Control. In *2017 IEEE International Symposium on*

High Performance Computer Architecture (HPCA).

[33] Amirhossein Mirhosseini, Mohammad Sadrosadati, Behnaz Soltani, Hamid Sarbazi-Azad, and Thomas F Wenisch. 2017. BiNoCHS: Bimodal network-on-chip for CPU-GPU heterogeneous systems. In *Proceedings of the Eleventh IEEE/ACM International Symposium on Networks-on-Chip*. ACM, 7.

[34] Sparsh Mittal and Jeffrey S. Vetter. 2014. A Survey of Methods for Analyzing and Improving GPU Energy Efficiency. *ACM Comput. Surv.* 47, 2, Article 19 (Aug. 2014), 23 pages. https://doi.org/10.1145/2636342

[35] Sparsh Mittal and Jeffrey S. Vetter. 2016. A Survey Of Architectural Approaches for Data Compression in Cache and Main Memory Systems. *IEEE Trans. Parallel Distrib. Syst.* 27, 5 (May 2016), 1524–1536. https://doi.org/10.1109/TPDS.2015.2435788

[36] NCSU. 2014. The FreePDK process design kit. Available: http://www.eda.ncsu.edu/wiki/FreePDK.

[37] Negin Nematollahi, Mohammad Sadrosadati, Hajar Falahati, Marzieh Barkhordar, and Hamid Sarbazi-Azad. 2018. Neda: Supporting direct inter-core neighbor data exchange in GPUs. *IEEE Computer Architecture Letters* 17, 2 (2018), 225–229.

[38] Nvidia. 2009. "Whitepaper: Nvidia's Next Generation CUDA Compute Architecture: Fermi".

[39] Nvidia. 2009. Nvidia CUDA SDK 2.3. [Online]. Available: http://developer.nvidia.com/cuda-toolkit-23-downloads.

[40] Nvidia. 2012. "Whitepaper: Nvidia's Next Generation CUDA Compute Architecture: KeplerGK110".

[41] Nvidia. 2014. "Whitepaper: Nvidia GeForce GTX 980".

[42] Yunho Oh, Myung Kuk Yoon, William J Song, and Won Woo Ro. 2018. FineReg: Fine-Grained Register File Management for Augmenting GPU Throughput. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 364–376.

[43] Jinpyo Park and Soo-Mook Moon. 2004. Optimistic Register Coalescing. *ACM Trans. Program. Lang. Syst.* 26, 4 (July 2004), 735–765.

[44] G. Pekhimenko, E. Bolotin, N. Vijaykumar, O. Mutlu, T. C. Mowry, and S. W. Keckler. 2016. A case for toggle-aware compression for GPU systems. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[45] Gennady Pekhimenko, Vivek Seshadri, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. 2012. Base-delta-immediate Compression: Practical Data Compression for On-chip Caches. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT '12)*.

[46] Mohammad Sadrosadati, Seyed Borna Ehsani, Hajar Falahati, Rachata Ausavarungnirun, Arash Tavakkol, Mojtaba Abaee, Lois Orosa, Yaohua Wang, Hamid Sarbazi-Azad, and Onur Mutlu. 2018. ITAP: Idle-Time-Aware Power Management for GPU Execution Units. *ACM TACO* (2018).

[47] Mohammad Sadrosadati, Amirhossein Mirhosseini, Seyed Borna Ehsani, Hamid Sarbazi-Azad, Mario Drumond, Babak Falsafi, Rachata Ausavarungnirun, and Onur Mutlu. 2018. LTRF: Enabling High-Capacity Register Files for GPUs via Hardware/Software Cooperative Register Prefetching. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 489–502.

[48] M. H. Santriaji and H. Hoffmann. 2016. GRAPE: Minimizing energy for GPU applications with performance requirements. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[49] A. Sethia, G. Dasika, M. Samadi, and S. Mahlke. 2013. APOGEE: Adaptive prefetching on GPUs for energy efficiency. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*.

[50] A. Sethia and S. Mahlke. 2014. Equalizer: Dynamic Tuning of GPU Resources for Efficient Execution. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*.

[51] A. Shafiee, M. Taassori, R. Balasubramonian, and A. Davis. 2014. MemZip: Exploring unconventional benefits from memory compression. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*.

[52] Premkishore Shivakumar and Norman P Jouppi. 2001. Cacti 3.0: An integrated cache timing, power, and area model. Technical Report 2001/2, Compaq Computer Corporation.

[53] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-Mei W Hwu. 2012. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing* 127 (2012).

[54] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. 2018. Mobilizing the micro-ops: Exploiting context sensitive decoding for security and energy efficiency. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 624–637.

[55] Nandita Vijaykumar, Gennady Pekhimenko, Adwait Jog, Abhishek Bhowmick, Rachata Ausavarungnirun, Chita Das, Mahmut Kandemir, Todd C. Mowry, and Onur Mutlu. 2015. A Case for Core-assisted Bottleneck Acceleration in GPUs: Enabling Flexible Data Compression with Assist Warps. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA '15)*.

[56] X. Wang and W. Zhang. 2017. GPU Register Packing: Dynamically Exploiting Narrow-Width Operands to Improve Performance. In *2017 IEEE Trustcom/BigDataSE/ICESS*.

[57] Daniel Wong, Nam S. Kim, and Murali Annavaram. 2016. Approximating warps with intra-warp operand value similarity. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[58] G. Wu, J. L. Greathouse, A. Lyashevsky, N. Jayasena, and D. Chiou. 2015. GPGPU performance and power estimation using machine learning. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*.

[59] Qiumin Xu and Murali Annavaram. 2014. Pattern Aware Scheduling and Power Gating for GPGPUs. In *Parallel Architectures and Compilation Techniques (PACT), 2014 23nd International Conference on*.

[60] Jun Yang, Youtao Zhang, and Rajiv Gupta. 2000. Frequent Value Compression in Data Caches. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO 33)*.

[61] Zahra Yarahmadi. 2016. *Study of the Bipartite Edge Frustration of Graphs*. Springer International Publishing, Cham, 249–267. https://doi.org/10.1007/978-3-319-31584-3_15

[62] Youtao Zhang, Jun Yang, and Rajiv Gupta. 2000. Frequent Value Locality and Value-centric Data Cache Design. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*.