Exploiting Direct Memory Operands in GPU Instructions

Ali Mohammadpur-Fard*, Sina Darabi†§, Hajar Falahati†¶, Negin Mahani‡¶†, and Hamid Sarbazi-Azad†*
*Department of Computer Engineering, Sharif University of Technology, Tehran, Iran
†School of Computer Science, Institute for Researches in Fundamental Sciences (IPM), Tehran, Iran
‡Department of Computer Engineering, Higher Education Complex of Zarand, Shahid Bahonar University, Kerman, Iran
§Faculty of Informatics, Universita della Svizzera Italiana (USI), Lugano, Switzerland
§Barcelona Supercomputing Center (BSC), Barcelona, Spain

***Abstract*—Modern GPUs are widely used for diverse applications, particularly data-parallel tasks like machine learning and scientific computing. However, their efficiency is hindered by architectural limitations, inherited from historical RISC processors, in handling memory loads causing high register file contention. We observe that a significant number (around 26%) of values present in the register file are typically used only once, contributing to more than 25% of the total register file bank conflicts, on average. This paper addresses the challenge of single-use memory values in the GPU register file (i.e. data values used only once) which wastes space and increases latency. To this end, we introduce a novel mechanism inspired by CISC architectures. It replaces single-use loads with direct memory operands in arithmetic operations. Our approach improves performance by 20% and reduces energy consumption by 18%, on average, with negligible (<1%) hardware overhead.**

***Index Terms*—GPGPU, RISC, Register File, CISC.**

I. Introduction

Graphics Processing Units (GPUs) are widely used in various application domains, i.e. General-purpose GPU (GPGPU) applications. GPGPU applications range from machine learning to scientific computing [1–6]. Data-parallel GPGPU applications benefit from running on GPUs due to their high computational density. However, GPGPU applications often do not utilize the full potential of GPUs due to the inherent limitations of the architecture, specifically, the way memory loads are performed by the GPU using the inherited ISA (instruction set architecture) of RISC (reduced instruction set computers) architectures.

Similar to RISC processors, GPUs use the register file to store two kinds of data values: (1) loaded values (fetched from memory by executing a memory load instruction), and (2) operand context of the current running threads [7]. By treating the register file as a data transfer intermediary between the various memory spaces, a large number of register file accesses are performed, and more bank conflicts can occur (leading to higher register file contention) due to the high Thread-Level Parallelism (TLP) in GPUs. Industry trends also attest to the importance of register file contention, by bypassing the register file for certain types of memory transfers. For instance, NVIDIA's Ampere and newer generations have introduced a new mechanism for moving data from Global Memory into the Shared Memory space directly, bypassing the register file [8].

To tackle this challenge, prior solutions including new scheduler designs [9], internal forwarding [10], register file caching [11], and register-sharing [12–14] mechanisms are introduced in GPUs. Certain scheduler designs can be employed to reduce the impact of the register file pressure by scheduling warps whose active instructions cause the fewest number of bank conflicts [9]. Some cache bypassing methods have been proposed to address the issue of low-locality data in caches [15, 16]. Similar mechanisms have also been introduced for bypassing the register file for consecutive reads/writes to the same register in a small instruction window, reducing execution latency [10]. A register cache [17] before the main register file further reduces contention and access latency through cache hits. The register-sharing mechanism combines registers into a physical one using methods like bit-packing [13], time-sharing [12], or de-duplication [14]. However, none of these methods directly solve the issue of single-use memory values in the register file, our main focus.

In this paper, we observe that a substantial portion of memory values (around 26%) stored in the register file could be eliminated, as they are exclusively utilized once (called single-use memory values in this paper). Moreover, we observe that these single-use memory values are the source of more than 25% of all register file bank conflicts, on average.

To this end, we propose a mechanism, called Direct Memory Operands (DeMO), for replacing single-use loads with direct memory operands; inspired by CISC (complex instruction set computer) architectures. To detect single-use load instructions, and represent them with our proposed format in the GPU instruction space, we introduce a new algorithm used in the compiler (for identifying these loads in live register ranges), as well as a new encoding. To avoid wasting the operand encoding space, we represent the new direct memory operands in arithmetic operations. The choice of adding direct memory operands only to arithmetic instructions is inspired by our observation that more than 50% of single-use loads are consumed by arithmetic instructions, with the second largest contributors being copies from one memory space to another (26%). Our cycle-accurate simulations show an average performance improvement of 20% (up to 74%) and GPU energy consumption reduction of 15% (up to 50%) over the standard implementation of GPGPU applications while having negligible area and compile-time overheads of less than 1% and 5%, respectively.

II. Background and Motivation

To gather the values of source operands and signal the readiness of a warp, an Operand Collector Unit (OCU) is

TABLE I: Baseline simulation configuration

SMs	Subcores/SM	RF banks/SM	OC banks/SM
72	4	8	8
Scheduler	**L1 dCache**	**L1 iCache**	**L2 Cache**
LRR	96KiB	128KiB	2MiB

assigned to a warp upon instruction issuance. This involves reading the source operands of the warps from the register file banks and buffering them in OCUs. These buffered units are utilized to space out register accesses in time, resolving bank conflicts without the need for re-accessing. To simplify the interconnection network, OCUs are designed as single-port buffers, which means multiple source operands must be serialized. Although multiple OCUs can simultaneously collect their allocated operands, all except one must also serialize their accesses in the event of a bank conflict. Once all the source operands are collected, an instruction is considered ready for execution. Upon completion of execution, the results are written back to the register file. This process is repeated for each instruction, resulting in a significant number of register file accesses, which contribute to a significant portion of the GPU's overall power consumption.

Due to the RISC-like instruction-set architecture, as well as the OCU design, performing a regular load instruction that reads from memory means the register file has to be accessed twice (once for the address and once for writing the result). Consequently, the number of register file accesses increases with the number of loads executed in a conventional GPU architecture. Coupled with the high TLP present in GPGPU applications, the increased number of register file accesses causes more bank conflicts and higher register file contention.

To provide some motivation, we simulate the execution of several baseline GPGPU applications in standard benchmark suites in Accel-Sim [18] on a Turing RTX 2060 GPU [19] (see Table I for configuration) and report the amount of register file contention (bank conflicts), the amount of single-use memory values in the register file, and the impact of such values on the register file contention. As Accel-Sim lacks complete support for newer GPU architectures, we implemented the Global-to-Shared Async Memory Copy mechanism present in the Ampere family [8] in the simulator separately. It should be noted that the simulations were also run on the Volta architecture; the newer architecture is reported here as similar trends were seen in both architectures.

Register file contention in GPUs occurs when multiple threads or instruction operands attempt to access different data elements that are stored in the same bank of the register file simultaneously. To study the reason behind the register file contention, we analyze register file values in terms of usage count. The utilization of registers for storing the result of a memory load operation in a RISC instruction set serves two purposes: (1) simplicity of the internal data path in the OCU, and (2) provision of a user-defined cache for future utilization of the loaded value. However, in scenarios where the loaded values are only used once, the allocation of a register and the subsequent read/write accesses become redundant and negatively impact the execution latency [20–24]. Moreover, the increased traffic in the register file due to unnecessary register file accesses can lead to additional bank conflicts, resulting in sub-optimal performance. We measure the number of reads performed on individual allocated registers to quantify the extent of such values. We identify single-use memory values using the following criteria: (1) the register must have been most recently written to by a load instruction, and (2) the register must have been read exactly once during its lifetime. Figure 1 illustrates that, on average, 26% of all register file accesses can be attributed to single-use memory values.

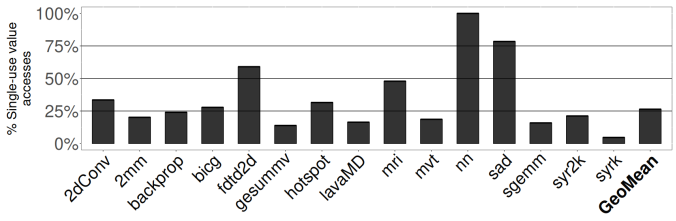

Fig. 1: Single-use memory value accesses in the register file.

Moreover, to analyze the impact of single-use memory values on register file contention, Figure 2 reports the number of single-use loads encountering a bank conflict normalized to the total register file bank conflicts. We observe that, on average, 25% of the total register file bank conflicts are due to single-use loads. Furthermore, to assess the impact of single-use memory values on GPU performance, we establish an Ideal scenario in which all single-use memory values bypass the register file entirely and incur no additional latency — by directly incorporating them into the target instruction. Figure 3 shows the performance improvement achieved in the aforementioned Ideal case, which results in a maximum performance gain of 75% and an average gain of 28%. Therefore, bypassing the register file for single-use loads can significantly reduce contention in the register file, improving GPU performance.

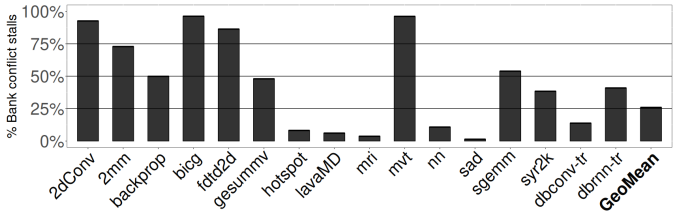

Fig. 2: Register file bank conflicts caused by single-use values.

III. Mechanism

We introduce DeMO, a new mechanism for reducing register file contention in GPUs. Unlike the conventional RISC-style memory access model, DeMO allows the instructions to perform direct memory access in place of regular operands, effectively eliminating the need for register file accesses when dealing with single-use values. This is done by rewriting eligible instructions into a new form, which can be interpreted by the hardware to allow the OCU to bypass the register

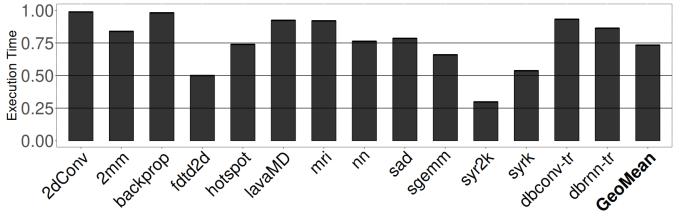

Fig. 3: Normalized execution time in the Ideal case.

Baseline PTX	Modified PTX	SASS pseudocode
1 ld.const.f32 %f1,[%rd1]	1' ld.const.f32 %f1,[%rd1]	1'' ld.const.f32 %f1,[%rd1]
2 ld.const.f32 %f2,[%rd1+4]	2' mul.ftz.f32 %f3 %c[%rd1+4] %f1	2'' ld.const.f32.d %v1,[%rd1+4]
3 mul.ftz.f32 %f3,%f2,%f1	3' fma.rn.ftz.f32 %f3,%f1 %f3,%f3	3'' mul.ftz.f32 %f3,%v1,%f1
4 fma.rn.ftz.f32 %f3,%f1 %f3,%f3		4'' fma.rn.ftz.f32 %f3,%f1 %f3,%f3

Fig. 4: An example of DeMO's logical changes.

file, leading to performance improvements, especially for applications that make heavy use of single-use values.

Figure 4 shows the baseline PTX, the modified PTX, as well as a pseudocode variant of the SASS code emitted as part of the compiler transformations applied by DeMO on a sample. In the baseline PTX, two memory loads and two arithmetic instructions are shown (`mul` and `fma`). Considering the first load instruction, DeMO does not rewrite the uses of the register `f1` as direct memory operands, because this register is used twice in the two instructions `3, 4`. However, DeMO rewrites the uses of register `f2` in the second load, because this register is used only once as an argument in instruction `3`, leading to instruction `2'`. The rewritten instruction `2'` is encoded as two SASS instructions `3''` and `4''` that respectively perform a direct memory load and a `mul` instruction that consumes the direct memory operand.

The proposed idea can be fully implemented in hardware or using a compiler-hardware co-design approach. Due to the high cost of fully implementing the proposed idea in hardware — including large instruction decode buffers required to accurately detect single-use loads – a compiler-hardware approach is chosen in DeMO. The compiler performs the transformations, ensuring that all the instructions are available to be examined. DeMO needs some minimal changes in the hardware to support directly loading an operand from memory and to bypass the register file for the fetched operand consumed by the immediately following instruction. The following subsections elaborate further on the compiler and hardware changes needed to support DeMO.

A. Compiler Support

We modify the compiler to support two main tasks in DeMO: (1) detecting instructions that are candidates to be rewritten, and (2) rewriting the detected instructions into the proposed form (see Figure 4 for the rewritten instructions). We use algorithms 1 and 2 for detecting single-use loads and rewriting them and their consumers, respectively.

Algorithm 1 examines an instruction window to pinpoint active register operands and returns a record mapping all registers containing single-use values to their loader and last consumer instructions (`Output`), initialized in line 3. DeMO performs all analyses statically at compile time, hence, the instruction window covers all possible instructions. However, reduction of the size of the instruction window can not cause Algorithms 1 and 2 to produce false positives, thanks to algorithmic safeguards explained below. Algorithm 1 iterates through all active instructions within the instruction window, creates a record mapping live registers to their usage counts, and the loader and last consumer instructions, which forms

Algorithm 1 Detecting single-use loads

```
1: function FindLoads(W)                                   ▷ Instruction window W
2:     R_active ← ∅                          ▷ Map register ⇒ usage count, loader, last consumer
3:     Output ← ∅                                          ▷ Map register ⇒ loader, consumer
4:     for each definitely-executed-instruction I in W do
5:         for each operand opr in I's input operands do
6:             if opr occurs in R_active then
7:                 Increment the usage count of opr in R_active
8:                 Update the last consumer of opr ← I in R_active
9:             end if
10:        end for
11:        for each operand opr in I's output operands do
12:            if opr ∈ R_active then
13:                if opr's usage count in R_active = 1 then
14:                    Output[opr] ← (opr's loader, opr's last consumer)
15:                end if
16:                Delete opr from R_active
17:            end if
18:            if I is a load instruction then
19:                R_active[opr] ← (0, I, 0)
20:            end if
21:        end for
22:    end for
23:    if no further instruction windows are present then
24:        for each reg ⇒ (usage count, loader, consumer) in R_active do
25:            if usage count = 1 then
26:                Output[reg] ← (loader, consumer)
27:            end if
28:        end for
29:    end if
30:    return Output
31: end function
```

Algorithm 2 Rewriting single-use loads and their consumers

```
1: procedure ReplaceLoadAndConsumer(W)                     ▷ Instruction window W
2:     M ← FindLoads(W)
3:     for each instruction I in W do
4:         if I ∈ M's consumers then
5:             Remove I from W
6:             Replace M's register in M's consumer with a direct memory operand
   equivalent to I
7:         end if
8:     end for
9: end procedure
```

the R_{active} structure initialized in line 2, and updates it while iterating through the instruction window (lines 4-22). During each iteration, the algorithm updates the usage count field of R_{active} by adding the number of register references to it (line 7) and the last consumer field by setting it to the current instruction (line 8). In the second loop over all output registers, when a register from R_{active} with a usage count of 1 is written to, the register is marked as single-use in `Output` (line 14), noting the loader and the consumer (lines 12-17). New entries are added to R_{active} when a load instruction is encountered (line 19), and they are removed when the register's lifetime ends (line 16). At the end of the instruction stream, all registers' lifetimes are considered terminated, allowing single-use loads to be detected in the final window (lines 23-29). Algorithm 2 first calls into Algorithm 1 to detect single-use loads. Then, iterating over all instructions in the instruction window (lines 3-8), it removes single-use loads (line 5) and replaces their consumer's operands with direct memory operands (line 6).

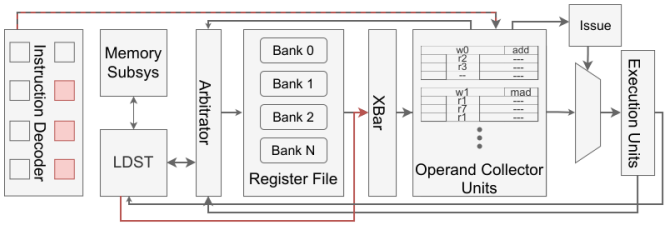

Fig. 5: The architectural datapath changes proposed by DeMO.

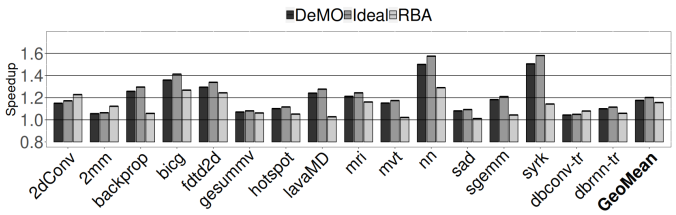

Fig. 6: Performance improvements in DeMO.

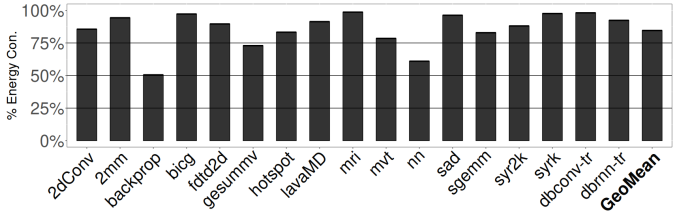

Fig. 7: Normalized GPU energy consumption in DeMO.

B. Architecture Support

To support direct memory operands, we (1) add a new instruction for every load instruction available in the baseline ISA, and (2) modify the architectural datapath, as shown in Figure 5. The new load instructions target a virtual register (representing the direct memory access operand) that allows the operand collector unit (OCU) allocated to the consumer instruction to identify active single-use loads and route them to the right buffer inside the OCU.

Figure 5 demonstrates the architectural datapath changes proposed by DeMO. These modifications allow for bypassing the intermediate register used for single-use loads. While simplistic, the changes in Figure 5 can be implemented by reusing existing datapaths in the OCU-ALU line, allowing for a much smaller routing and area footprint. The only required architectural changes are: (1) modifications to the Instruction Decoder, to decode the newly added instructions, and (2) connecting OCUs to the Load/Store units (LDST) through a direct line. As such, DeMO can be easily extended to support new kinds of load instructions in the future.

IV. Experimental Evaluation

The Accel-Sim framework [18] is used to implement the method with the RTX 2060 baseline configuration (as detailed in Section II), and the Accel-Wattch plugin [25] is used to measure the energy consumption. The assessments use select GPGPU applications from the Rodinia 3.1 [26], Parboil [27], Lonestargpu-2.0 [28], Polybench [29], and Deepebench [30] benchmark suites. To measure the hardware overheads, we utilize Synopsys Design Compiler to synthesize the Hardware Description Language model of all added components within DeMO for the NanGate 45-nm open cell library.

Performance and Energy Analysis: The performance results of all designs are shown in Figures 6. The measures given for "RBA" correspond to an implementation of only the register file bank conflict reduction measures proposed in [9], as the other scheduling mechanisms are purely orthogonal to DeMO. We make two key observations: (1) an average performance improvement of 20% can be seen, and (2) certain applications (e.g. `syrk`) show a large performance improvement despite a lower number of reduced register file accesses. The performance enhancement in both observations can be attributed primarily to two factors: (1) reduction in register file bank conflicts stemming from decreased contention, and (2) reduction in memory load latency in low-TLP applications.

As depicted in Figure 7, an average improvement of 18% can be seen in GPU energy consumption. This reduction directly results from the decreased execution time of the applications, and the lower number of register file accesses.

Overhead Analysis: Our proposed design adds new load instructions (one for each of `ld`, `ldc`, `ldg`, `ldl`, and `lds` in Turing), which affects the ISA and the Instruction Decode Units; as well as modifying the OCU datapaths with a direct link to Load-Store units (<1% hardware overhead). Our design also adds a compiler pass that performs the necessary instruction rewrites; this pass can be efficiently (`O(N)` time and memory) implemented as a buffered instruction stream processor, yielding a small overhead in compile times (<5%).

V. Conclusion

In this paper, we proposed DeMO to improve GPU performance by replacing single-use memory loads with direct memory operands which decreases total accesses to the register files and its contention. DeMO follows a compiler-hardware co-design approach to eliminate single-use loads and rewrite their consumers to use direct memory access operands. Evaluation results showed that DeMO improves GPU performance by 20% and reduces GPU energy consumption by 18%, on average, with negligible hardware and compile-time overhead.

References

[1] Nematollahi et. al. Efficient nearest-neighbor data sharing in gpus. *TACO*, 2020.
[2] Darabi et. al. Nura: A framework for supporting non-uniform resource accesses in gpus. *SIGMETRICS*, 2022.
[3] Darabi et. al. Morpheus: Extending the last level cache capacity in gpu systems using idle gpu core resources. In *MICRO*, 2022.
[4] Darabi et. al. OSM: Off-chip shared memory for GPUs. *TPDS*, 2022.
[5] Falahati et. al. Data-Aware compression of neural networks. *IEEE CAL*, 2021.
[6] Sadrosadati et. al. ITAP: Idle-time-aware power management for GPU execution units. *ACM TACO*, 2019.
[7] Gómez-Luna et. al. Chapter 13 - cuda dynamic parallelism. In *Programming Massively Parallel Processors (Third Edition)*. 2017.
[8] NVIDIA Corp. Nvidia ampere tuning guide. Technical report, 2023.
[9] Barnes et. al. Mitigating gpu core partitioning performance effects. In *HPCA'23*.
[10] Asghari et. al. Breathing operand windows to exploit bypassing. In *MICRO'20*.
[11] Jing et. al. Cache-emulated register file: An integrated on-chip memory architecture for high performance gpgpus. In *MICRO*, 2016.
[12] Khorasani et. al. Regmutex: Inter-warp gpu register time-sharing. In *ISCA*, 2018.
[13] Wang et. al. Gpu register packing. In *Trustcom/BigDataSE/ICESS*, 2017.
[14] Jeon et. al. Gpu register file virtualization. In *MICRO*, 2015.
[15] Chao et. al. Locality-driven dynamic gpu cache bypassing. In *ICS'29*, 2015.
[16] Xie et. al. An efficient compiler framework for cache bypassing on gpus. In *ICCAD'13*.
[17] Sadrosadati et. al. Highly concurrent latency-tolerant register files for GPUs. *ACM TOCS*, 2021.
[18] Khairy et. al. Accel-sim: An extensible simulation framework for validated gpu modeling. In *ISCA*, 2020.
[19] NVIDIA Corp. Nvidia turing gpu architecture. Technical report, 2018.
[20] Nematollahi et. al. Neda: Supporting direct inter-core neighbor data exchange in gpus. *CAL*, 17(2):225–229, 2018.
[21] Mostofi et. al. Snake: A variable-length chain-based prefetching for gpus. In *MICRO*, pages 728–741, 2023.
[22] Falahati et. al. Power-efficient prefetching on GPGPUs. *Supercomputing*, 2015.
[23] Falahati et. al. ISP: Using idle SMs in hardware-based prefetching. In *CADS*, 2013.
[24] Keshtegar et. al. Cluster-based approach for improving graphics processing unit performance by inter streaming multiprocessors locality. *IET Comput. & Digit. Tech.*, 2015.
[25] Kandiah et. al. Accelwattch: A power modeling framework for modern gpus. In *MICRO*, 2021.
[26] Che et. al. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC'9*.
[27] Stratton et. al. Parboil: A revised benchmark suite for scientific and commercial throughput computing. 2012.
[28] Burtscher et. al. A quantitative study of irregular programs on gpus. In *IISWC'12*.
[29] Grauer-Gray et. al. Auto-tuning a high-level language targeted to gpu codes. In *Innovative Parallel Computing*, 2012.
[30] Baidu Research. deepbench. https://github.com/baidu-research/DeepBench, 2017.