Locality-aware GPU Register File

Hyeran Jeon *Member, IEEE*,
Hodjat Asghari Esfeden,
Nael B. Abu-Ghazaleh *Member, IEEE*,
Daniel Wong *Member, IEEE*, Sindhuja Elango

Abstract—In many emerging applications such as deep learning, large data set is essential to generate reliable solutions. In these big data workloads, memory latency and bandwidth are the main performance bottlenecks. In this paper, we propose a locality-aware GPU register file that enables data sharing for memory-intensive big data workloads on GPUs without relying on small on-chip memories. We exploit two types of data sharing patterns commonly found from the big data workloads and have warps opportunistically share data in physical registers instead of issuing memory loads separately and storing the same data redundantly in their registers as well as small shared memory. With an extended register file mapping mechanism, our proposed design enables warps to share data by simply mapping to the same physical registers or reconstructing from the data in the register file already. The proposed sharing not only reduces the memory transactions but also further decreases the register file usage. The spared registers make rooms for applying orthogonal optimizations for energy and performance improvement. Our evaluation on two deep learning workloads and matrixMul show that the proposed locality-aware GPU register file achieves over 2× speedup and saves register space up to 57%.

Index Terms—Matrix Operations, Convolution Neural Network, GPU

1 Introduction

In big data era, one of the most critical computing problems is to speedup memory accesses. To deliver huge data requested by massively parallel threads, big data workloads such as deep learning are accelerated on many-core processors such as GPUs that are equipped with multiple high-bandwidth memory controllers as well as various on-chip memories. However, many big data workloads inherently have data sharing features. For example, large matrix operations such as matrix multiply and dot product are one of the most commonly used algorithms in big data workloads. In matrix multiply, any two neighboring threads that are assigned to the adjacent output matrix entries in a row use the same row of the first input matrix. Also, the core algorithm of convolutional neural network (CNN), which is one of the most successful deep learning algorithms is dot-product operation. In a CNN, neighboring neurons compute dot-product of sliding sub-windows of the input data, where sub-windows overlap significantly.

Figure 1 shows an example data sharing across neighboring neurons in a convolution (conv) layer computation. The colored entries of the output matrix are the conv results

- *H. Jeon is with the San Jose State University, San Jose, CA 95192. E-mail: hyeran.jeon@sjsu.edu.*
- *H. A. Esfeden, N. B. Abu-Ghazaleh, and D. Wong are with the University of California Riverside, Riverside, CA 92521. E-mail: hodjat.asghari@email.ucr.edu, {daniel.wong, nael}@ucr.edu.*
- *S. Elango was a student at San Jose State University and now with Synopsis Inc., Mountain View, CA 94043. E-mail: sindhuja@synopsys.com.*

Manuscript submitted: 24-Oct-2019. Manuscript accepted: 15-Nov-2019. Final manuscript received: 22-Nov-2019

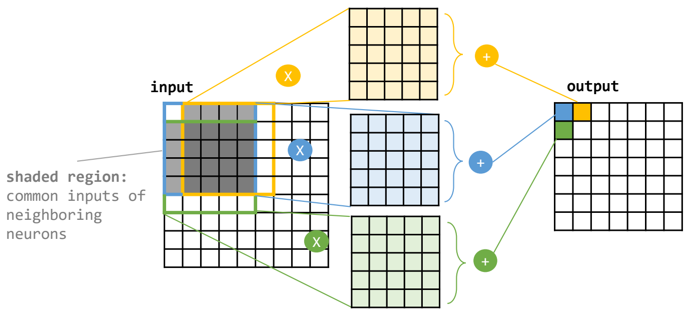
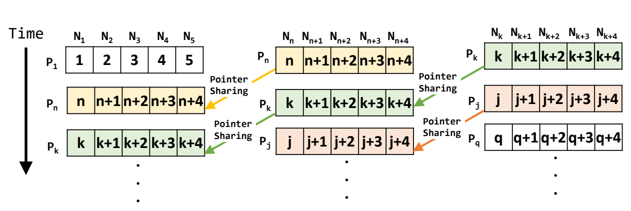


Fig. 1: Input data sharing among three neurons that are vertically and horizontally neighboring in a conv layer

of three neurons where blue and yellow neurons are horizontal neighbors and blue and green neurons are vertical neighbors. Each neuron uses a sub-set of data of the input matrix, which is highlighted with colored border lines. The blue neuron takes the data in the first 5×5 matrix region, runs dot-products, and stores the result to the output entry. Likely, the green and yellow neurons each takes a matrix from one row below and one column after the blue neuron's input matrix. The data in the shaded regions that overlap between any two input regions are used by both neurons. In this example, 20 out of 25 input data of each neuron are shared between any two neighboring neurons and 16 of them are used by all three neurons. In the typical CNN computing, as individual neurons executions are considered as independent, each neuron issues 25 memory reads even when the overlapping data are somewhere in the on-chip memory already. These redundant memory accesses lead to an excessive usage of memory resources. Even when many of the data can be cached as they are accessed in the similar time windows, due to the limited size of L1 cache, L2 cache should be also excessively accessed. Given that interconnection between L1 and L2 is one of the critical performance bottlenecks of GPUs [1], it is not desirable that all neurons independently access memory. Also, shared memory can be also used for reducing the memory access latency. But, due to the limited size, the code should be carefully designed to have each CTA to load only a small block of data over multiple iterations as in blocked matrix multiply. Also, using shared memory adds a burden of loading data from global memory to shared memory and then register file.

In this paper, we propose to reduce the redundant memory accesses by sharing data directly in the register file without any help from L1 or shared memory. Instead of having the three neurons in Figure 1 to store their 25 input data separately, our approach stores the overlapping 20 data to a common set of registers. A neuron accesses memory only when the data is not already in the register file. This approach not only reduces the redundant memory accesses but also decreases the register file usage because the three neurons share one physical copy of register for each of the overlapping data rather than having them in their private register separately. If data sharing is done in the other on-chip memories such as shared memory, four copies of each data should be stored in the SM: a copy in the shared memory and three copies in the register file for the three neurons. Given the massive number of neurons used for each conv layer, significant resource waste is ex-


(a) Perfect sharing among vertically neighboring neurons

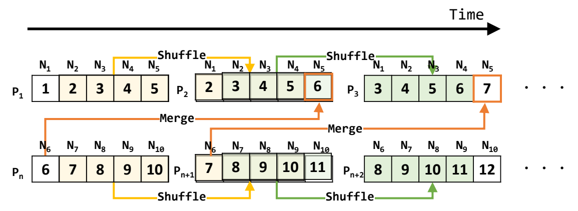

(b) Partial sharing among horizontally neighboring neurons

Fig. 2: Proposed Data Sharing (an example of CNN)

pected if using other on-chip memories for data sharing. The registers and the other on-chip memories that are saved by our approach may be used for improving the performance further by running more CTAs [2]. The larger size and the lower typical utilization also makes the GPU register file more favorable for the data sharing over the other on-chip memories [3]. Therefore, we leverage large register file space to support more threads by preserving common data longer without worrying about cache thrashing, insufficient shared memory space, or a sophisticatedly designed software.

On-chip inter-thread data sharing have been explored by several studies [4], [5], [6]. However, many of them either used specialized data flow architectures where data sharing is easier via direct communication channels among processing elements, or focused on time-series parameters sharing where excessive concurrent accesses do not need to be considered. WIR [7] leveraged physical register sharing to skip instructions. As WIR focused on reducing arithmetic operation executions for a better energy efficiency, WIR has a more complex design that consists of hash and instruction meta information tables. Also, the register reuse is allowed only when the warp-unit register has a perfectly identical data. Our approach focuses on memory access overhead and hence exploits common memory access patterns that enables warps to share data for both perfect- and partial-matching cases with a simpler address mapping table. The state-of-the-art deep learning libraries use register file and shared memory to speedup the data access latency [8]. However, software-level data sharing leads to an excessive register file usage with redundant data copies because current GPUs do not allow inter-warp register sharing. In this paper, we extend the architecture-to-physical register mapping in the architecture level to enable data sharing across warps.

2 Data Reuse in Register File

Warps may share either perfectly-matching data or partially-matching data. We explain how our approach supports both cases by using a conv layer example.

2.1 Perfect Sharing

Any two vertically neighboring neurons have commonly used input data as illustrated as a shaded region between the inputs of the blue and the green neurons in Figure 1. The entire contents of each row in the overlapping region are shared by the two neurons, which enables perfect sharing. As each neuron is assigned to a GPU thread in typical CNN implementations, vertically neighboring neurons are likely to be in different warps. For example, if the blue neuron is thread 0 of warp 0, the green neuron is thread 0 of warp N. Figure 2a shows the register updates of perfect sharing in a warp unit, where we assume that each warp has only five threads for simplicity. Each five-element array is a warp-unit register where each entry is a register of a thread in the same warp. The three warp-unit registers in the same row are of the three groups of neurons that are vertically neighboring (i.e., the warp having N_1 to N_5 is the one that has the blue neuron, the N_{th} warp consisting of N_n to N_{n+4} is the one that has the green neuron, and the K_{th} warp is the one that is one row below the N_{th} warp). The second row of the first warp is identical with the first row of the N_{th} warp. Likely, the third row of the first warp is the second row of N_{th} warp as well as the first row of K_{th} warp.

These vertically overlapping rows can be directly shared by mapping the same physical register pointer to the architectural registers of the neighboring warps. For example, if the N_{th} warp loaded its first row to a physical register P_n, the first warp can get its second row values by simply mapping its destination register of the load instruction to P_n without needing to access memory. Likely, once one of the three warps loads the values in P_k, the other two warps can get these data by simply mapping their registers to P_k. The perfect sharing can be also found among horizontally neighboring warps in matrix multiply. For example, if two 128×128 matrices are multiplied, four warps that are assigned to the same row will use the same input row of the first matrix that can be shared by using physical register mapping. The perfect sharing is light weight because it only requires to map an architectural register to an existing physical register instead of a memory load. However, it is possible only when warps use exactly the same data. For partially-matching data, we use partial sharing.

2.2 Partial Sharing

The blue and yellow neurons in Figure 1 show the partial sharing between neighboring neurons. The horizontally neighboring neurons are likely to be processed by the neighboring threads in the same warp. For example, if the blue neuron is assigned to the thread 0 of warp 0, the yellow neuron is mapped to the thread 1 of warp 0. For each data load, these threads read data that are next to each other with a given stride distance. Figure 2b shows the register updates by this load pattern in a warp unit when stride is 1. The three arrays in each row show the first three data that each of the five neurons load from the memory. For example, N_1 loads 1, 2, and 3 (the first data in each array) as its first three data and stores them to the first 32-bit entry of the warp-unit registers P_1, P_2, and P_3, respectively. For each load operation, the values of the warp-unit register are shuffled down by one of the previous load result. For example, N_1 uses 2 as the second data, which is the first data of N_2 and so on. This operation can be represented with NVIDIA

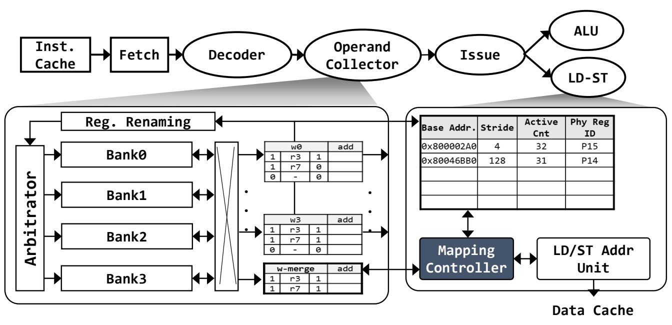


Fig. 3: Architectural Modification

CUDA Shuffle instruction, $P_2 = \_\_shfl\_down(P_1,\ 1,\ 5);$. If the destination register value of each load is kept until the second load, the majority of the data (four out of five each time) can be reused without issuing another load operation.

We still have one data that should be newly loaded each time (6 and 7 in the second and the third loads, respectively). These values can be fetched from neighboring warp's registers. The second row of Figure 2b shows the register updates of the neurons 6 to 10, which are grouped to the second warp. As warps are interleavingly scheduled in GPUs, after the first warp loads the data 1 to 5, the second warp loads the data 6 to 10. When the first warp is to load the second values, the data 6 is already in a register P_n. Therefore, the second warp-unit register values of the first warp can be constructed by merging the shuffled four values (2 to 5) and the first value of the second warp (6). By running two simple logical operations on the existing register values, we can skip memory loads for partially-matching data.

3 Architectural Modification

To share data in a register across multiple warps, it is essential to decouple the architectural registers from the physical registers. As the dynamic architectural-to-physical register mapping has been already evaluated by several studies [3], we assume that our baseline register file already incorporates virtual register mapping. The only difference between the baseline and our mapping is that one physical register id may appear in multiple entries of the register mapping table to be mapped to multiple architectural registers, which doesn't need any modification in the register mapping mechanism.

Figure 3 shows the modified architecture. The components with bold outline and dark color are newly added. To indicate the physical register that holds the shared data, an *Address Mapping Table* is added in the load-store unit. Once a load address is calculated and the address is bound to the global memory, the address mapping table is looked up. If the same address is not in the table already, the address and the destination physical register are recorded to the table and the load is issued. The data sharing in our proposed approach is done in warp unit. Thus, each entry of this table contains the information of a warp-unit load accesses. As threads in a warp typically access consecutive addresses with a fixed stride, we record the base address (the address of the first thread), the stride, and the active lane count rather than storing addresses of all the threads.

Once a memory address is recorded in the address mapping table, any following load instructions look up the table and retrieve the data from the physical register that is mapped the same address and skip the load executions. The destination architectural register's entry in the register mapping table is updated with the physical register id mapped to the load address in the address mapping table. This register sharing is done in hardware, without any help from compiler. The table lookup and register update is controlled by *Mapping Controller*. An entry of the address mapping table is deleted when the corresponding memory address is newly written or when there is no more mapped architectural registers. When a store instruction is issued to the load-store unit, the address is checked from the address mapping table and the corresponding entry is deleted. Note that to enforce memory consistency, we exclude data declared as $volatile$ from the register sharing. In GPU, it is rare but possible that the CTAs communicate through global memory. To reflect the remote updates that cannot be detected by the local operations, the compiler is slightly modified to include the volatile variables to the renaming-exempted registers [3] such that load operations are not skipped for them. Also, when an architectural register becomes dead and the mapped physical register is released, the mapping controller checks the register mapping table if there is any more architectural register mapped to the same physical register. If there is no more mapped register, the associated address mapping table entry is deleted.

The address mapping table is sufficient for supporting perfect sharing. However, to support partial sharing that needs to merge two registers, we add one dedicated operand collector slot as marked with $w - merge$ in Figure 3. This new slot loads two registers that contain subsets of the requested warp-unit register data. These two registers can be found by checking the address mapping table. For each global memory load instruction, if mapping controller finds that no entry in the address mapping table can cover the requested data, it finds up to two entries that together cover the requested warp-unit data. If there are no two entries that can support all the requested data, the load is issued to the memory. Otherwise, the two registers are requested to the operand collector logic and the load execution is skipped. Once two register values are ready, the mapping controller uses them to construct the requested data. For this merging process, we design a small ALU that runs bit-level SHIFT operations for shuffling and an OR operation for merging. The merged value is written to the load instruction's destination register at the writeback stage.

The perfect sharing does not add any performance overhead because it only requires one address mapping table lookup and a register renaming table update. If the corresponding entry is found from the address mapping table, it does not even require register writeback because the destination register is mapped to a physical register that already has the requested value. The partial sharing takes longer time than the perfect sharing because two registers should be read from the register file and shuffle and merge operations should be done on them. Thus, we run partial sharing only upon L1 cache misses. L1 cache miss can be easily detected without extra logic because the mapping controller runs in the load-store unit, which probes L1 cache for memory operations and operates miss handling upon cache misses [9].

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/LCA.2019.2959298, IEEE Computer Architecture Letters

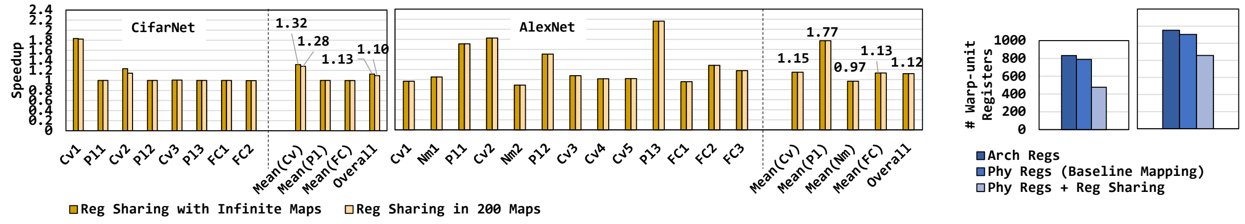


Fig. 4: Speedup and Register Usage of CNNs. (Cv : Conv, Pl : Pool, Nm : Norm, FC : Fully Connected)

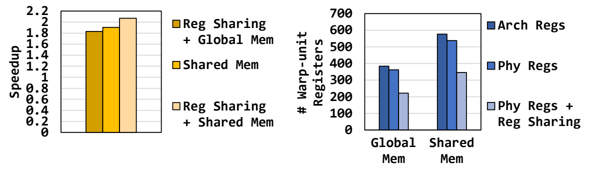


Fig. 5: Speedup and Register Usage of 128×128 MatrixMul

4 Evaluation

The idea is implemented in GPGPU-Sim with baseline virtual register mapping [3]. Two-level warp scheduler and 128KB per-SM register files are used. We evaluated performance and register file usage while running two CNNs of Tango DNN benchmark suite [10] and matrixMul.

4.1 Performance

Figure 4 shows the evaluation results of CifarNet and AlexNet. We measured the performance with infinite mappings and with 200 mappings only (address mapping table size becomes 1KB). With 1KB mapping table, per-layer and end-to-end performance showed up to 116% and 12% speedup, respectively. In CifarNet, the earlier layers showed higher improvement due to the larger feature map size. Note that smaller feature maps break individual warp-unit loads into multiple stride groups and require multiple entries in the address mapping table. To support more data-intensive layers with a simpler design, we support only one stride group per load access. In AlexNet, pooling layers showed even higher speedup than conv layers because the 3D input structure of conv layers impedes data sharing by increasing the stride length, while its overlapping-based max pooling provides higher data sharing opportunity. The speedup was mainly sourced from the reduced global memory accesses, where a total of 36% and 20% reduction was observed in CifarNet and AlexNet, respectively. We also measured the breakdown of the partial and perfect sharing, where the partial sharing contribution of earlier to later conv layers is decreased from 56%, 29% to 16%, and 24% to 1% in CifarNet and AlexNet, respectively (the contribution of perfect sharing is 100% - that of partial sharing), while pooling layers were mostly sped by partial sharing. The blue graphs of Figure 4 show the max utilization of warp-unit registers during the execution. Our approach saved 57% and 26% register space in CifarNet and AlexNet, which makes 44KB and 37KB extra register file space available.

For the matrixMul, we evaluated an algorithm without any optimization and a blocked-matrixMul using shared memory to understand the impact of the register sharing over the existing optimizations. Our approach without any optimization showed 83% speedup as plotted with the darkest yellow in Figure 5 while software optimization (medium yellow) shows only 8% further speedup. Regarding the register usage, the optimized matrixMul uses 50% more registers than unoptimized one, while our design reduced almost 40% register usage, which spares 17KB and 24KB register space. With these spared registers, register sharing allowed to run more CTAs and achieved 10% further speedup over the optimized code as shown in the lightest yellow bar.

4.2 Area

Our design adds an operand collector slot, an address mapping table, and a mapping controller. An operand collector slot adds insignificant overhead as GPUs have 16 operand collector slots per SM already. Each address mapping table entry consists of a total of 38 bits where base address, stride, and active count are 24 bits, 9 bits, and 5 bits, respectively, which are set by the load address range and stride size of DNN workloads and matrixMul. The address mapping table size is limited to 1KB and the register renaming table in the baseline is 1KB.

5 Conclusion

We propose a locality-aware GPU register file that exploits two common data sharing patterns of big data workloads. The proposed mechanism improves performance and reduces register usage with which the performance and energy can be further improved by applying orthogonal optimizations.

References

[1] G. Koo, H. Jeon, Z. Liu, N. S. Kim, and M. Annavaram, "CTA-Aware Prefetching and Scheduling for GPU," in *IPDPS*, 2018.
[2] Y. Oh, M. K. Yoon, W. J. Song, and W. W. Ro, "FineReg: Fine-Grained Register File Management for Augmenting GPU Throughput," in *MICRO*, 2018.
[3] H. Jeon, G. S. Ravi, N. S. Kim, and M. Annavaram, "GPGPU Register File Virtualization," in *MICRO*, 2015.
[4] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: Efficient Inference Engine on Compressed Deep Neural Network," in *ISCA*, 2016, pp. 243–254.
[5] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A Spatial Architecture for Energy-efficient Dataflow for Convolutional Neural Networks," in *ISCA*, June 2016.
[6] F. Khorasani, H. A. Esfeden, N. Abu-Ghazaleh, and V. Sarkar, "In-Register Parameter Caching for Dynamic Neural Nets with Virtual Persistent Processor Specialization," in *MICRO*, October 2018.
[7] K. Kim and W. W. Ro, "WIR: Warp Instruction Reuse to Minimize Repeated Computations in GPUs," in *HPCA*, 2018.
[8] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cuDNN: Efficient Primitives for Deep Learning," in *arXiv:1410.0759*, 2014.
[9] X. Zhu, R. Wernsman, and J. Zambreno, "Improving first level cache efficiency for gpus using dynamic line protection," in *in ICPP*, 2018.
[10] A. Karki, C. P. Keshava, S. M. Shivakumar, J. Skow, G. M. Hegde, and H. Jeon, "Detailed Characterization of Deep Neural Networks on GPUs and FPGAs," in *GPGPU*, 2019.