CS 534 Final Project Report

Qingyuan Liu, Yushan Li, Qingzhe Liu, Pranamya Jain, Weiqi Zhang

ABSTRACT

Graphics Processing Units(GPUs) have already been developed into powerful and useful accelerators for the general-purpose and parallel-required works, which may not restrict in only their original graphics domain. The baseline for each GPU architecture is to mainly rely on the Single Instruction Multiple Data(SIMD) execution model, which can have multiple processing elements to execute the same instruction across different data elements in parallel. In most of the cases, we are encountering cases that require tons of vector and matrix operations, which can be beneficial from data parallelism. However, in the real-world GPU, we will also deal with a high frequency of scalar operations, where all the active lanes of the SIMD instruction operate on the same input value. In this case, using SIMD pipelines to run the scalar instructions can lead to inefficient use of resources in GPU and introduce more unnecessary power consumptions because regular SIMD models treat truly paralell vector instructions and redundant scalar instructions identically.

To address this inefficiency, the prior works [1] have introduced a scalar-vector GPU architecture to separate two execution paths. It maintains the original vector pipeline and adds a new scalar pipeline to incorporate scalar instructions, and thus reduces the redundancy of recalculating and hence reduces the power. In this project, we followed their work to rebuild the scalar-vector GPU architecture. After that, we evaluate our work on the benchmark provided by GPGPU-sim and on Rodinia [2], which are two widely used GPU benchmarks, to show the performance increment and power reduction.

1 INTRODUCTION

Graphics Processing Units (GPUs) employ a Single Instruction Multiple Data (SIMD) execution model, which allows multiple processing elements to execute the same instruction across multiple data lanes. However, this execution model does not differentiate between instructions that operate on the same data across all lanes and those that do not. As a result, redundant computations occur when scalar instructions (those that have identical data across lanes) are treated as standard SIMD operations, leading to inefficient execution and increased power consumption.

The paper "Balancing Scalar and Vector Execution on GPU Architectures" [1] proposes a scalar-vector execution model that introduces dedicated scalar execution units to handle these redundant computations efficiently. By adding a separate scalar execution pipeline, the scalar instructions are completely separated from those real vector instructions, thus reducing energy consumption and improving performance. However, the paper does not take into account the additional area costs associated with adding both scalar ALUs and scalar memory components, which is potentially infeasible for real implementation.

In our project, we will implement the scalar-vector execution model and explore ways to come up with a practical and implementable structure. Our ideas can be divided into two parts. First, we aim to reduce the scalar components to make the design more feasible for implementation (e.g., by removing scalar ALUs or eliminating scalar memory components). This will inevitably lead to some performance reduction. The next step is to introduce the scalar structure at a finer granularity, where data similarity is evaluated in halves. Through this modified approach, we seek to balance efficiency improvements with practical hardware costs.

2 BACKGROUND

Today, modern GPUs rely on data parallelism to achieve high throughput. Figure 1 shows the big picture of the structure of NVIDIA GTX480, which is the main GPU we run simulations with. It contains 15 Streaming Multiprocessors(SMs), where each SM has 32 warps. The scheduler will automatically distribute our instructions into each thread inside each warp to maximize the occupancy, and hence to improve the IPC – especially for the vector or matrix computations. While this architecture delivered a step-function increase in throughput, its vector-only pipeline will perform any warp-uniform ("scalar") instruction multiple times—such as loop counters, address calculations, or uniform branches—across all 32 lanes. This will lead to huge redundancy in terms of instruction fetch rate and register traffics. In addition to the performance, the dynamic power would also be increased due to more "waste" instructions.

Another GPU programming model is Open Computing Language (OpenCL). It is a heterogeneous programming framework used for running programs on CUDA-powered GPUs. Figure 2 shows the basic structure of an OpenCL context and how the host and the device ends interact. The programs objects are piles of kernels, which will be executed on the devices later. The memory objects on the right consist of buffers, images, and pipes, which are the data we read from or write to on the devices. The events allows programmers to track when kernel launches, executions, or memory transfers complete. The command queues retrieve each instruction for a specific device. Finally, the device queues are fed with

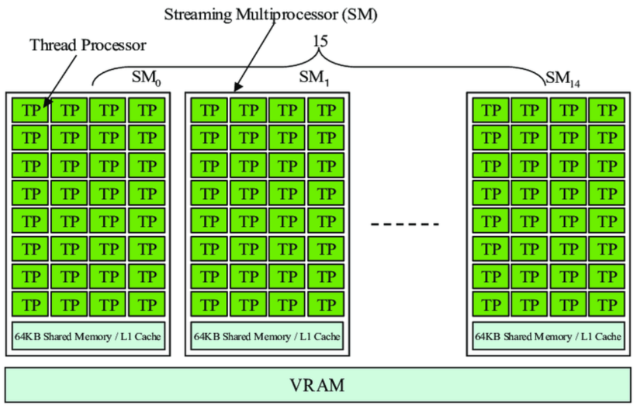

Figure 1: GTX480 CUDA Architecture with 15 SMs and 32 TPs for each SM [3]

instructions stored in the command queues in order, execute them, and write back to memory objects.

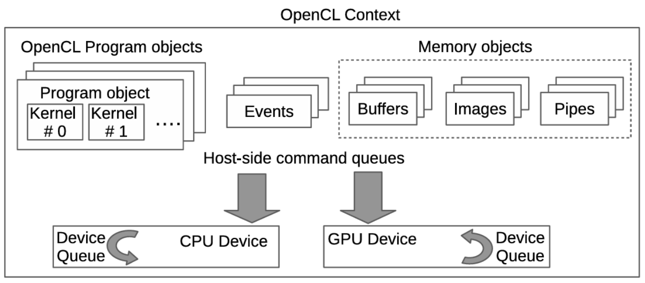

Figure 2: OpenCL programming model [4]

In order to further improve the throughput for GPU architecture, the scalar instructions can be extracted from the vector heavy computations and feed those instructions to the scalar ALUs. Figure 3 shows the prototype of this thought – we omit some detailed structures for scalar+vector GPU such as the interconnections, scalar cache and so on. If we feed, for example, ADD operation into 32 lane warp, it will recognized as the vector operation and duplicating the ADD operation for 32 times. For only one instruction, we have already created 31 unnecessary computations and waste both the available threads and power. With the additional scalar ALU included, ADD operation can be recognized as the scalar computation and directly feed into scalar part. Hence, we will not have 31 more redundant operations and the throughput for each of the scalar operation encountered by the GPU will be boosted for about 32 times faster ideally if we can neglect the latency of the further control logic introduced to determine the types of the operation and to transfer the corresponding type on correct path.

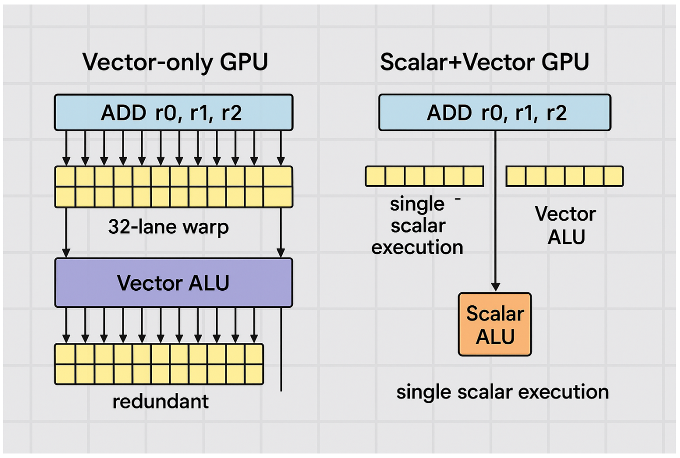

Figure 3: Comparison of vector-only GPU vs. Scalar-vector GPU

3 METHODOLOGY

3.1 Overview

We first present an overview of our new GPU architecture. Figure 4 shows all the blocks that are required for building the scalar-vector GPU architecture. We keep most of the blocks used in the vector computations and add or modify the scalar blocks to mimic the similar structures.

To first determine whether the instructions received by the operand collector are scalar, we need the scalar detector unit. The basic logic of this unit is to check whether all the threads in the operand collector are same. If they are same, the additional bit will become 1 to indicate that the operation is scalar and vice versa. To save the area, this additional bit will be carried from operand collector to dispatch unit so that we do not need the scalar detector for each unit.

The dispatch block will be extended to include one more bit that determines whether the instruction is scalar or vector. The output of the dispatch block will also be broadcast to four – two for scalar units and two for vector units. The additional bit acts as an arbitrator to determine which path the instruction should be dispatched on. The scalar ALU unit will receive the scalar instruction and directly perform one-time computation and then extend it to 32 bits for the output, and the scalar memory unit will also load or store the data correspondingly.

For the Writeback unit, we need to increase the input ports from two to four to handle all the outputs coming from scalar ALU, scalar Mem, vector ALU and vector Mem. In addition to that, the extended bit to determine the type of operation is still carried with the instruction. The output of the Writeback unit will be sent to either scalar register or vector register depending on the scalar detector bit.

For our project, we emphasize on designing the scalar units for GPU architecture without changing the warp scheduler and adding the optimization for the dispatch units – our dispatch will fire the instructions as long as they have been determined as scalar or vector and are ready.

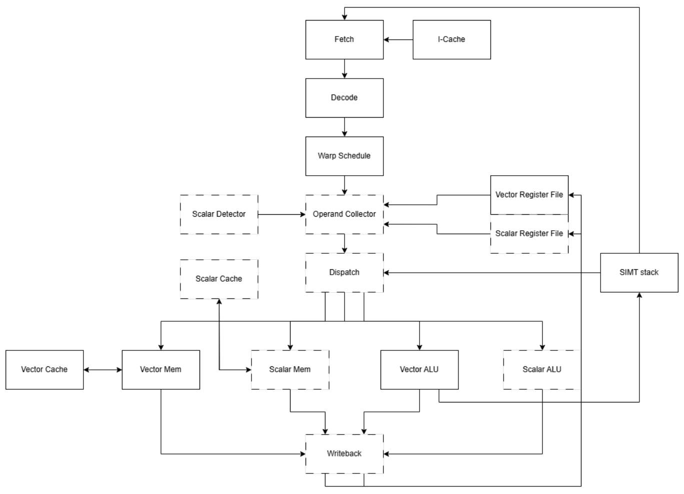

Figure 4: The block diagrams of the scalar-vector GPU architecture. Blocks with dashed lines mean they are newly added or have been modified

3.2 Implementation

Our implementation of scalar/vector logic and design is based on GPGPU-Sim 3.2.2 [6] with the configurations of an NVIDIA GTX480 calibrated.

3.2.1 ***Scalar Detector.*** The scalar instruction detection mechanism is to identify ALU instructions which can be executed using a scalar datapath. Our architecture supports scalar execution only for regular arithmetic ALU instructions: control-flow instructions (e.g. BRA) and memory-related operations (e.g. LD, ST, TEX) are explicitly excluded from scalar execution.

The detection process takes place at the functional execution stage in the simulator because the actually operand values are collected only in this stage. The detection process operates as follows:

1. Instruction Type Filtering
   As described above, instructions that are not arithmetic ALUs are filtered out at this stage. Only these are eligible for scalar execution.
2. Source Operand Analysis
   An eligible scalar instructions should have all source operands being scalar.
   - Immediate Value Check: If the operand is an immediate value (i.e., constant), it is scalar.
   - Relocation Table Check: A relocation table is maintained to record which registers have most recently been written by scalar instructions or vector ones. If the operand's corresponding register is marked scalar in the table, we consider it scalar.
   - Vector Register File Check: If the operand resides in the vector register file (VRF), we fetch its value across all active threads in the warp. If all values are identical, we consider the operand scalar.
3. Final Scalar Decision
   If and only if all source operands are classified as scalar according to the above checks, a 1-bit flag ($scalar\_flag$) in the instruction is marked 1 for following scalar process.

This layered approach makes it possible to quickly bypass expensive per-thread data comparisons when possible, while still falling back on full operand comparison when needed.

3.2.2 ***Operand Collectors.*** To support efficient scalar instruction pipeline, we introduce a lightweight scalar operand collection mechanism distinct from the existing vector operand collector system. Two dedicated scalar operand collector units are added to enable parallel operand collection for multiple warps concurrently issuing scalar instructions. In the baseline vector design, operand collection involves contentions both on VRF and collector units. This requires arbitration to resolve bank conflicts and resource unavailability, which may delay operand readiness by multiple cycles. In contrast, the scalar operand collector design adopts a significantly simplified model:

1. No Bank Conflicts
   Scalar operands are assumed to reside in a scalar register file that is either single-ported or small enough to avoid bank conflicts. Thus, no port or bank arbitration is necessary.
2. Single-Cycle Read Latency
   Once a scalar instruction is assigned to a scalar operand collector, all source operands are read in one cycle. These operands are considered ready without delay.

As soon as the operands are ready, the instructions are forwarded to the simulator's *m_issue_port* to become eligible for issuing to the corresponding ALU units.

3.2.3 ***Scalar ALU Units.*** In the baseline GPGPU-Sim architecture, the functional operation (i.e. the real operation like add, subtraction) and the pipeline simulation are separated. The functional simulation is placed actually very early before the operand collector part. Then, the results values are carried along with the instruction to going through the pipeline simulation. The typical execution flow for vector ALU instructions is as follows:

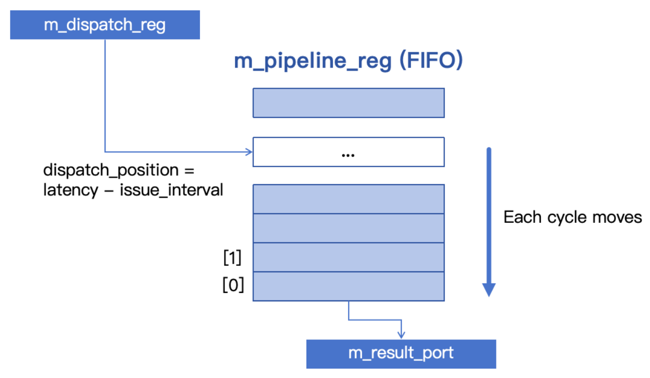

Figure 5: FIFO structure in the simulator for pipeline simulation.

1. Operand Collection
   Once all source operands are marked ready by the operand collector, the instruction is dispatched to `m_result_port`, which is connected to the pipeline register stage `OC_EX_SP`.
2. Issue
   From OC_EX_SP, the instruction is issued to the available SP functional unit. GPGPU-Sim models multiple SP units that support concurrent instruction execution across warps. The pipeline is simulated as a FIFO in the simulator, with entry 0 connected to m_result_port which will then used to write back and complete the whole process. The issue of an instruction means putting the instruction to be executed with its calculated results to an entry in the pipeline FIFO. The entry where an instruction is going to be started depends on the delay and the dispatch interval of its opcode. Figure 5 shows a structure of this pipeline FIFO.
3. Execute
   Every cycle, the contents of the FIFO shift forward by one stage, simulating the advancement of instructions through the pipeline.
4. Writeback
   Once execution completes (i.e. the instruction and results finally reaches m_result_port), results are written back to the vector register file, subject to possible port or bank contention. The instruction then retires from the pipeline.

For scalar SP unit, we use the similar class as the vector SP unit. Two additional scalar SP units are added. Following description in the paper, the clock used to feed the scalar SP unit is 3x faster than the normal one, in order to logically provide a 1:1 ratio of SIMD units to scalar units.

For the functional execution part, unlike vector instructions which loop through every active thread and operate on all of them, scalar instructions only operate once on the first active lane. The results are also stored only on the first active lane until write back.

Similar to the operand collection, the writeback of the scalar instruction only taken contentions of resources, not including bank conflicts.

3.2.4 *Scalar Register File.*

In the baseline GPGPU-Sim architecture, the vector register file (VRF) is modeled at the per-thread level. That is, each thread maintains its own independent set of architectural registers. This design reflects the SIMT (Single Instruction, Multiple Threads) programming model, where every thread executes logically in parallel and has its own private register space. In the simulator, this is implemented as a map from register identifiers to register values inside each thread context (ptx_thread_info). When an instruction accesses a register during the functional simulation process, it loop through each active thread and access the corresponding register sets to get the operand values for calculation.

This per-thread register organization allows precise modeling of divergent behavior and enables accurate tracking of individual thread state, but it can lead to redundancy when multiple threads in a warp operate on the same values. To optimize execution of warp-uniform scalar instructions, we introduce a scalar register file (SRF) that is organized at the warp level, rather than per-thread. Each warp maintains a shared SRF instance that stores scalar values accessible to all threads within that warp. The SRF is accessed by scalar instructions, where all active threads in the warp share the same operand values. This allows the simulator to avoid redundant register accesses and reduce register storage pressure.

In our implementation, the SRF is modeled as an array of maps, one map per warp. Each maps symbolic register identifiers (symbol*) to scalar register values (ptx_reg_t) within a warp. 6 shows the difference between VRF and SRF. When a scalar instruction is issued, each operand marked with scalar in the relocation table is read from the SRF, and the result is written back to the SRF if current instruction is scalar. This organization reflects a hardware realization where scalar registers are implemented once per warp, yielding better energy efficiency and reduced memory traffic, particularly in scalar-heavy code regions.

3.2.5 *Relocation Table.*

To support accurate and dynamic tracking of scalar register values within a hybrid scalar-vector execution architecture, we introduce a relocation table. The relocation table serves as a lightweight bookkeeping structure that records which registers are currently stored

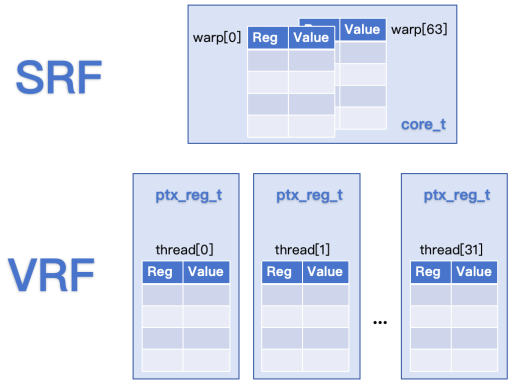

Figure 6: FIFO structure in the simulator for pipeline simulation.

in SRF or VRF, enabling components such as the scalar detector and functional executions to efficiently decide where to obtain the real values or decide whether this operand is scalar or not.

The relocation table is implemented as a std::map<const symbol*, bool> per warp. It is updated at the end of each instruction at the functional execution stage:

- If a scalar instruction writes to a register, update the register symbol in the map of the corresponding warp to true, indicating accesses to this operand will be directed to SRF.
- If a vector instruction writes to a register, update the register symbol in the map of the corresponding warp to false, meaning the following access to this operand should be in VRF.

This mechanism introduces minimal overhead but enables the simulator to support a dynamic, fine-grained scalar/vector execution model with high accuracy and flexibility.

4 EVALUATION

We used GPGPU-Sim 3.2.2 to simulate our design and evaluate performance. Since we do not have tools to extract the hardware and test the area, this metric will not be considered in this project. We used GPUWattch to analyze power.

For the benchmark, we use Rodinia 3.0 that both include an approximately similar number of scalar instructions and vector instructions. The selected benchmarks include BFS (Breadth-First Search), LavaMD (Molecular Dynamics simulation), and HotSpot (Thermal simulation) among the few from Rodinia.

We have collected several metrics for this implementation. The first is the instructions per cycle (IPC), which is the main metric to examine that the modified implementation could maintain a relatively high throughput compared to the original implementation. The second is the total average power consumption, which would indicate whether our implementation could decrease the power usage compared to the original implementation.

5 RESULTS

5.1 IPC

Figure 7 shows a comparison of the IPC metric between the baseline and our modified scalar-vector implementation. While a marginal IPC increase of up to 8.35% was observed across all benchmarks (with an average increase of 3.05%). This result is mainly due to the decrease in the redundant instructions and hence decrease the simulation time for each benchmark. The absolute results for IPC are shown in Table 1.

BM	B Inst/s	B Cyc/s	B IPC	VS Inst/s	VS Cyc/s	VS IPC
bfs	144761	2939	49.26	134112	2513	53.37
backprop	691505	1190	581.10	662006	1125	588.45
b+tree	663450	2027	327.31	389600	1174	331.86
Hotspot	882327	25243	34.95	530850	14166	37.47
lud	468442	957	489.49	362613	740	490.02
pathfinder	813438	1565	519.77	657053	1207	544.37
nw	189159	4798	39.42	144535	3619	39.94
lavaMD	175368	1922	91.24	135749	1487	91.29

Table 1: IPC comparison: Baseline (B) vs Vector-Scalar (VS) Arch.

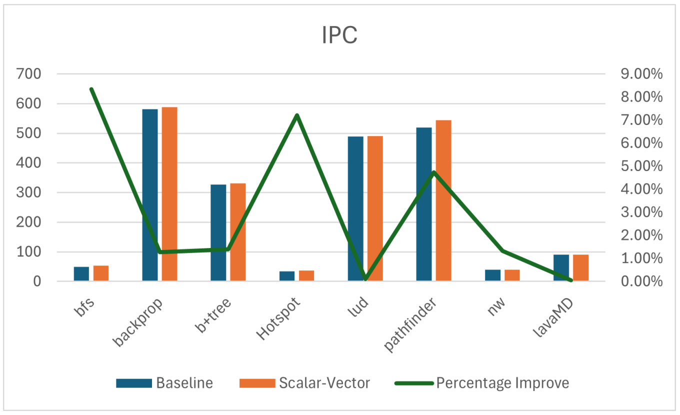

Figure 7: IPC of baseline and scalar-vector architecture. Higher IPC indicates better performance.

5.2 Total Average Power

Figure 8 shows the comparison of the total average power (in Watts) between the baseline and our implementation. Seen from the figure, the average power is reduced for all sets of benchmarks, ranging from 3.76% to 12.37%, with an average of 9.59%. This indicates that our implementation successfully reduced the power consumption by executing the vector instructions with scalar ALUs. The absolute results for Power are shown in Table 2.

Benchmark	B Power (W)	VS Power (W)	Gain (W)	% Improve
bfs	74.35	69.08	5.27	7.08%
backprop	135.25	119.28	15.97	11.81%
b+tree	67.36	59.11	8.24	12.24%
Hotspot	36.79	35.41	1.38	3.76%
lud	89.93	80.06	9.87	10.97%
pathfinder	85.66	75.13	10.53	12.29%
nw	95.38	89.49	5.89	6.17%
lavaMD	49.24	43.15	6.09	12.37%

Table 2: Power usage: Baseline (B) vs Vector-Scalar (VS) Arch.

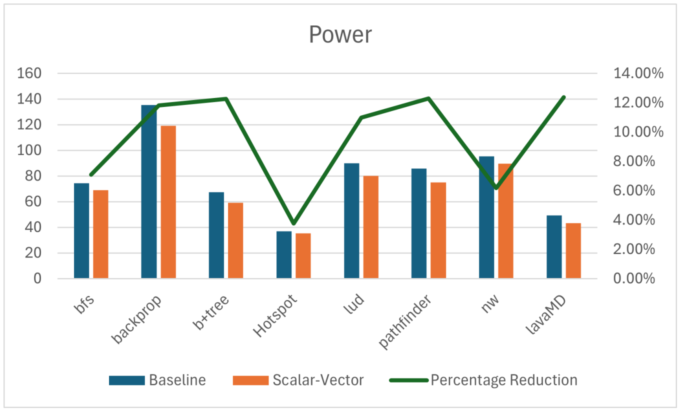

Figure 8: Power of baseline and scalar-vector architecture. Lower power indicates better performance.

6 RELATED WORK

In this very section, we briefly discuss some works related to our current project progress and future directions.

6.1 Current Work

From paper “Balancing Scalar and Vector Execution on GPU Architectures” [1], the original design of scalar-vector GPU architecture is to have a scalar detector attached right after the operands collector to record if each operand is scalar or not. However, this structure needs to wait until all the operands have been determined as scalar or not in order to dispatch the instructions into corresponding computational units, which takes some time. Hence, we want to modify this scalar detector to be separated by two parts – the first 16 threads in the first half of the warp and the second 16 threads in the second half of the warp. In this case, we might need two scalar detectors and an additional combining logic for two 16 threads to become one 32 threads output , but we can dispatch them with twice the speed and there will be less latency for us to wait until all the bits go to 1. In addition, from the paper “Characterizing Scalar Opportunities in GPGPU Applications” [5], the evaluation of GPU executions provides the result that nearly 51% of the instructions are scalar. Therefore, with double the number of scalar detectors and separate the instructions into two parts, we can hopefully increase both the speed of the system to determine whether it is scalar or vector and the power efficiency of the GPU.

6.2 Future Work

Moreover, the paper "Exploiting uniform vector instructions for GPGPU performance, energy efficiency, and opportunistic reliability enhancement" [7] shows besides redundancy within a uniform vector, different vectors can also have identical values. Addressing redundancy in different vectors could be a potential future direction or extension for our project. Last but definitely not least, as the paper "Power-efficient computing for compute-intensive GPGPU applications" states [8], increasing the number of compute resources and/or their frequency can increase the peak performance of GPUs, but it can also significantly increase power consumption. The register file bandwidth in GPUs is a critical resource that is optimized for 32-bit instruction operands. However, many operands require considerably fewer bits for accurate representation and computations. One power-efficient technique is to have a sliced GPU architecture that improves performance of the GPU by dual-issuing instructions to two 16-bit execution slices. This may also be incorporated into our GPU architecture.

6.3 Limitations

6.3.1 *Scalar Relocation Under Divergent Control Flow.*

One known limitation in our current scalar register file (SRF) design arises under divergent control flow.

In cases where only a subset of threads in a warp writes to a register (e.g., inside a conditional branch), the relocation table may not reflect the fact that the register now holds non-uniform values. If the table still marks the register as scalar, subsequent reads may incorrectly fetch the value from the scalar register file, even though the register no longer holds a uniform value across the warp.

Consider a simplified scenario as follows:

```
@%p0 bra ELSE;

IF:
    add.u32 %r7, %r7, 1; // only 1 thread
    bra END;

ELSE:
    nop;

END:
    ... = %r7;           // all threads
```

Listing 1: PTX code illustrating scalar inconsistency under divergence

If %r7 is modified only by 1 thread (due to conditional branching), but the relocation table is not updated to reflect this partial overwrite because the instruction is still considered a scalar one inside this conditional branching. The simulator may incorrectly read %r7 from the SRF for all threads after returning to the main pass, leading to functional errors.

To resolve such kind of problem, one way is to set up compiler limitations. For example, we add additional bits in the ISA to indicate that this instruction is ineligible for a scalar optimization. [?] The compiler will take up the responsibility to recognize such kinds of error scenarios and set those bits up. During the scalar detector part, it only needs to add another filter for instructions with these bits up. This approach needs limited hardware cost but will significantly increase compiler's complexity and affect its performance.

Another effective way is to invalidate the scalar relocation entry whenever a register is partially written. That is to say when only a subset of the warp's threads write to a given register, the register is no longer warp-uniform, and thus its scalar designation becomes invalid. In such cases, the relocation table entry will be set to false, and the old value in the SRF will be synchronized to all threads in VRF before updating new values. This requires more efforts on hardware design and increases data movement between SRF and VRF which may result in worse energy and speed performance.

6.3.2 ***Simple Schedule and Dispatch Strategy.*** One big issue with the current design is that we only modify the total number of ports and add scalar units for the GPU. This design is able to determine whether the operand collected is scalar or vector and send it to correct ALU or memory units. However, we did not make modifications on the scheduler and dispatch optimizations. With only the scalar-vector GPU structure without additional optimizations, it can waste time on sticking to the long-latency instructions – especially the vector instructions that can occupy most of the available threads and while leaving some threads idle. With some further greedy algorithm implemented for handling the mix of scalar and vector instructions, we can utilize the available source of GPU more efficiently and improve the IPC and decrease the power consumption further.[11]

6.3.3 ***Area.*** Even though we did not synthesize the design, we would expect that the area of the scalar-vector GPU will increase some amount especially with the later optimized warp scheduler, optimized dispatch units and scoreboard.[9] We need to reevaluate the overall performance by considering the area, power, cost and IPC to see if that is worthy to add on the new traits in the current GPU systems.

7 CONCLUSION

As a powerful and useful accelerator, GPUs make use of the SIMD execution model, which makes use of parallelism to handle vector instructions. However, in the real world, many workloads also involve scalar instructions, and resulting in a waste in the GPU resources. To eliminate this issue, prior work introduced a vector-scalar GPU architecture. In our project, we re-implemented the GPU architecture they have proposed, and removed some of the scalar components to reduce the additional chip area cost and ease the implementation. The performance of our modified scalar-vector architecture was evaluated with [BENCHMARKS]. Our implementation achieved an average of 3.05 % of increment in IPC and 9.59% of reduction in power consumption. Despite the promising results from experiment, there are limitations in our project. Lack of optimization in warp scheduling might result in a long stalling latency, unable to modify compiler result in failure in an internal floating-point exception in some edge cases, and absence of hardware synthesize cannot estimate the additional area usage in the project.

REFERENCES

[1] Z. Chen and D. Kaeli, "Balancing Scalar and Vector Execution on GPU Architectures," *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Chicago, IL, USA, 2016, pp. 973-982, doi: 10.1109/IPDPS.2016.74.

[2] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proc. of the IEEE International Symposium on Workload Characterization (IISWC)*, Austin, TX, USA, 2009, pp. 44-54, doi: 10.1109/IISWC.2009.5306797.

[3] S. Tsutsui and N. Fujimoto, "ACO with tabu search on GPUS for fast solution of the QAP," *Natural Computing Series*, 2013, 10.1007/978-3-642-37959-8_9.

[4] D. Kaeli, P. Mistry, D. Schaa, and D. P. Zhang, *Heterogeneous Computing with OpenCL 2.0.*, Waltham, MA, USA: Elsevier Science, 2015.

[5] Z. Chen, D. Kaeli and N. Rubin, "Characterizing scalar opportunities in GPGPU applications," *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Austin, TX, USA,

2013, pp. 225-234, doi: 10.1109/ISPASS.2013.6557173.

[6] T. M. Aamodt, W. W.L. Fung, and T. H. Hetherington, *GPGPU-Sim Manual*, 3.1.1 ed., 2017. [Online]. Available: http://www.gpgpu-sim.org/manual/

[7] P. Xiang, Y. Yang, M. Mantor, N. Rubin, L.-R. Hsu, and H. Zhou, "Exploiting uniform vector instructions for GPGPU performance, energy efficiency, and opportunistic reliability enhancement," in *Proc. Int. Conf. Supercomputing (ICS)*, 2013, pp. 433–442, doi: https://doi.org/10.1145/2464996.2465022.

[8] S. Gilani, N. S. Kim, and M. Schulte, "Power-efficient computing for compute-intensive GPGPU applications," *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Minneapolis, MN, USA, 2012, pp. 445-446.

[9] D Salonikidis, D. E. Manolakis, "Statistical Performance Analysis in a GPU," *2022 IEEE International Conference on Networking, Architecture and Storage (NAS)*

[10] G.-Y. Lueh, K. Chen, G. Chen, J. Fuentes, W.-Y. Chen, F. Fu, H. Jiang, H. Li, and D. Rhee, "C-for-Metal: High Performance SIMD Programming on Intel GPUs," *2021 IEEE ACM Int. Symp. Code Generation and Optimization (CGO)*

[11] I. Chaturvedi, B. R. Godala, Y. Wu, Z. Xu, K. Iliakis, P.-E. Eleftherakis, S. Xydis, D. Soudris, T. Sorensen, S. Campanoni, T. M. Aamodt, and D. I. August, "GhOST: a GPU Out-of-Order Scheduling Technique for Stall Reduction", *ISCA, 2024*