Engineering and Technology Journal for Research and Innovation (ETJRI)
ISSN 2581-8678, Volume V, Issue I, Dec 2022

Register Liveness Analysis to Enhance Thread Occupancy

Sandeep Gupta[1], Kuldeep Narayan Tripathi[2*]
[1]*Assistant Professor, Dept. of CSE, Amity University Madhya Pradesh Gwalior, sgupta@gwa.amity.edu*
[2*]*Assistant Professor, Dept. of CSE, Amity University Madhya Pradesh Gwalior, kntripathi@gwa.amity.edu*

Abstract—**The hardware complexity can be reduced when the register requirement is statically calculated and exclusively dedicated to executing the thread blocks in lifetime duration. This considers the maximum number of live registers at any given point in the thread block execution. However, the time at which all the requested registers are needed constitutes a tiny fraction of the kernel program. This static allocation, therefore, results in under-utilization of the register file. This paper focuses on the various general purposes of parallel computing programs to determine the extent of the register file's under-utilization due to the static allocation by performing the liveness analysis. Previous studies have already found that the register file is the main thread occupancy limiter in the GPUs, which gives an insight that if the registers are used effectively by having some dynamic allocation scheme, then the thread occupancy could be increased manifold. This paper has proposed a static pool allocation count using a hybrid approach compared with AlexNet kernel models. The implemented hybrid model has generated a lesser count than the AlexNet kernel model.**

Keywords—**GPGPU, register file, warp, resident threads.**

I. INTRODUCTION

Graphics Processing Units (GPUs) are widely adopted across various domains due to their massive thread-level parallelism (TLP). The TLP present in the GPUs is limited by the number of resident threads, which depends on the available resources in the GPUs such as registers, shared memory, etc. [1]. Since the allocation of threads to an SM is at the thread block granularity, some resources may not be used up entirely and hence remain underutilized. When the resources limit the applications, the number of resident thread blocks gets limited. As a result, the TLP in the SM also gets limited [2].

This work is a data-driven approach. We analyze the register usage for various benchmarks to see how static register allocation at the thread block scheduling results in wastage of resources[3].

Hence, it becomes a bottleneck for the throughput by restricting the number of resident threads. We have performed our register liveness analysis using nvdisasm and nsight tools provided by CUDA 10.2.89 for the benchmarks [4]. As per our observation, we think that instead of having a static register allocation mechanism at the scheduled time for a thread block, it is better to have a hybrid approach of allocating few registers at the scheduled time (from static physical register pool) and allocating more registers later on demand (from the dynamic physical register pool). For programs that require a large number of registers for execution, the hybrid register allocation scheme will enable a higher number of concurrent warps to be resident in the SM,

leading to higher thread occupancy in the GPU at any instant of time [5].

In this work, we presented an analysis of register liveness to enhance the thread occupancy to address the issues mentioned above. We proposed a synergistic compiler-hardware mechanism to make use of the hybrid register allocation scheme. This paper makes the following contributions:

- We analyzed the register liveness and observed that a hybrid approach to the registers' allocation/de-allocation would be better than the static and exclusive policy used today.
- We propose a hybrid approach that divides the register file into two pools: A static register pool and a dynamic register pool.
- We offer a synergistic compiler-hardware mechanism to make use of the hybrid register allocation scheme.

The rest of the paper is organized as follows: Section 2 expresses the motivation for a hybrid GPU register file approach. Section 3 describes some background knowledge that was needed for this paper analysis. Section 4 describes our proposed theoretical solution, elaborating on the tasks to be performed on the compiler end and the tasks performed on the hardware end (when SM schedules the thread block). Section 5 presents the approach and methodology, and Section 6 presents about Observation and Analysis, Section 7 is the conclusion.

II. MOTIVATION

To improve the TLP and resource utilization, we need to launch additional thread blocks in each SM. The extra thread blocks can help in improving the throughput by hiding long execution latency cycles. The register file is, by far, the single biggest occupancy limiter for threads (47%) observed in workloads [1]. The main culprit is the de facto GPU resource allocation policy. GPU architectures allot a CTA's register resources when the architecture schedules the CTA. The allotment

conservatively pro- visions for registers that statically might be needed but may not be dynamically used. The architecture frees the entire set of registers allotted to a warp only when all that warp threads have exited. By profiling applications for the number of simultaneously live registers, we can get an upper bound for how wasteful the de facto GPU policy is [6]. This analysis would guide us to come up with a better register allocation/de-allocation scheme.

III. BACKGROUND RESEARCH

A. Register Liveness Analysis

The first step in register allocation is to perform liveness analysis on the programs Control Flow Graph. Liveness Analysis is a data-flow analysis technique used by compilers to calculate the live range of every register. A live range is defined as a write to a register followed by all the uses of that register until the next write. A variable is live at a certain point if it holds a value that is needed in the future [5].

Optimum resource partition for competing kernels on one SM. Elastic kernel [25][7] enhances register and shared memory occupancy through simultaneous kernels released on a single SM. To improve resource use, the SMK [8] offers a dynamic sharing process for competing kernels.

Engineering and Technology Journal for Research and Innovation (ETJRI)
ISSN 2581-8678, Volume IV, Issue II, Jun 2022

Kernel Merge [9] and Space Multiplexing [10] studied how to use competing kernels to improve GPU resources and overall performance. To expand the use of GPU, Lee et al.[11]also exploit mixed competing kernels. In this work, we aim for a single kernel to use resources. For particular scenarios, some works note the GPU sub-utilization problem. To avoid warps at barriers for far too long, SAWS [12] and BAWS [13] are proposing barrier awareness scheduling policies. CAWS [14] and CAWA [15] predict and speed up lagging warts so that the thread block (TB) can end more quickly. Warp LEVEL [16]points out the subtilization of space and time resources due to the allocation of TB resources and proposes the use of warp resources to improve GPU resource utilization. Although the efficient TLP can improve at barriers and TB terminations, the maximum number that is permitted to be issued to the GPU is limited.

Zorua [17] uses GPU context virtualization to provide portability to programming and increase TLP levels. Zorua differs from our approach with two key differences. First, Zorua assigns/distributes resources on the chip at the phase limits. Whereas, when a warp/TB suffers from a long latent operation, our approach allocates resources. Second, Zorua pours out the register and shared storage overwritten files into the global storage, while our work uses the replacement on-chip resources to change the context much faster. Gebhart et al. [18] proposed to unify L1 D-cache, common memory, and record file to enhance GPU use of on-chip resources. We also benefit from increasing occupancy of registered or memory-limited shared applications. However, the unified design calls for extensive hardware changes and software support. They must also pay the overhead for distribution because different kernels have various resource requirements.

Some previous works [19][12][20][21] use a GPU pre-emption context switching. Like Lin et al. [19] proposes, we use lightness analyses and compression registrations to reduce the size of the context. In this work, however, we use spare resources for saving the spilled contexts to allow context shifting much faster.Dublish et al. [22]conduct a bottleneck analysis of GPU memory hierarchy at various levels, including the L1/L2 and DRAM caches. Neither deals, however, with the performance impact of the L1-L2 interconnection. Majumdar et al. [13] are looking at the scalability of the computer and memory bandwidth GPU kernels.

IV. MAIN IDEA

Based on the analysis of the register liveness graphs for the various benchmarks, we are proposing a hybrid approach for the allocation of the physical registers. The idea is to divide the physical register file into two pools: Static and Dynamic pool. At the time of scheduling a thread block, SM would first allocate the registers from the Static pool base‘ “1 d on the information provided to it by the compiler from the static register liveness analysis and then at the execution time the thread blocks would make the request for more registers from the dynamic register pool set. Early releasing the dead registers can enable the CTA scheduler to launch more thread blocks to the same SM even before the on-going thread-block completes its execution, resulting in an increase of thread occupancy. The approach can be used to develop models which can be used for wireless sensor networks [23], IoT [24] and many more.

A. Compile Time

At compile time, after performing the liveness analysis, the compiler can add indicator/instruction at the start of the kernel SASS code, which will indicate how many registers, should be allocated at

Engineering and Technology Journal for Research and Innovation (ETJRI)
ISSN 2581-8678, Volume IV, Issue II, Jun 2022

the time of scheduling the thread block itself from the static pool. Compiler would also add information by doing instrumentation at the start of the basic blocks which would need more than static number of live registers for computation; this would enable the SM to allocate registers for the thread block from the dynamic register pool at the start of a new basic block. Compiler can also insert indicators at the start of the basic blocks for the early release of the registers which are no longer going to be needed in the kernel execution. The early register release is based on the idea proposed in the paper [1], where it defines a new term Register Working Set to determine the point after which quite a lot of registers are not needed anymore in the kernel execution.

B. Execution Time

With the introduction of dynamic register pool, the SM would be required to be more intelligent as it needs to manage and keep track of dynamic pool registers. SM needs to maintain a register mapping table for each warp from architected registers to physical registers. Need to maintain a free list of registers for the unmapped physical registers. Therefore, the hardware complexity is going to increase but it’s a trade-off which would enhance the thread occupancy by better utilization of the registers[25].

V. APPROACH AND METHODOLOGY

In order to collect the data regarding the register liveness, we made use of the nvdisasm and Nsight Compute tool to generate Control-Flow-Graph[26] and number of live registers vs. instructions graph for the kernels[27]. We obtained both the static register liveness values, solely based on CFG, which ignores the fact that how many times a particular basic block is actually getting executed for a kernel, but this would provide us with some insight on how much percentage of instructions have live register count below a particular value. If the majority of the instructions (for instance more than 50%) could be satisfied with a certain live register count, then at the time of scheduling the thread block, SM would allocate that many registers from the static pool, leaving the thread block to demand more registers during execution from the dynamic register pool [28].

A. Tools Used [6]

1) Nvdisasm [4]: The NVIDIA CUDA disassembler. It can extract information from standalone Cubin files and present them in human readable format. nvdisasm also does control flow analysis to annotatejump/branch targets. Nvdisasm is capable of showing the register liveness range information. For each line of CUDA assembly, it displays whether a given device register was assigned, accessed, live or re- assigned. It shows the total number of registers used.

2) Nsight [6]: NvidiaNsight Compute is a kernel profiler for CUDA applications. After configuring which kernels to profile, which metrics to collect, etc., the application is run under the profiler. The collected profile can be stored in a file with nsight-cuprof-report extension. Using Nsight, we can launch or attach to a target application to obtain the desired metrics.

```
        .global         _Z9vectorAddPKfS0_Pfi
        .type           _Z9vectorAddPKfS0_Pfi,@function
        .size           _Z9vectorAddPKfS0_Pfi,(.L_24 - _Z9vectorAddPKfS0_Pfi)
        .other          _Z9vectorAddPKfS0_Pfi,@"STO_CUDA_ENTRY STV_DEFAULT"
_Z9vectorAddPKfS0_Pfi:
.text._Z9vectorAddPKfS0_Pfi:
        /*0008*/                   MOV R1, c[0x0][0x44];
        /*0010*/                   S2R R0, SR_CTAID.X;
        /*0018*/                   S2R R3, SR_TID.X;
        /*0020*/                   IMAD R0, R0, c[0x0][0x28], R3;
        /*0028*/                   ISETP.GE.AND P0, PT, R0, c[0x0][0x158], PT;
        /*0030*/               @P0 EXIT;
        /*0038*/                   ISCADD R4.CC, R0, c[0x0][0x140], 0x2;
        /*0048*/                   MOV32I R7, 0x4;
        /*0050*/                   IMAD.HI.X R5, R0, R7, c[0x0][0x144];
        /*0058*/                   ISCADD R2.CC, R0, c[0x0][0x148], 0x2;
        /*0060*/                   LD.E R4, [R4];
        /*0068*/                   IMAD.HI.X R3, R0, R7, c[0x0][0x14c];
        /*0070*/                   LD.E R2, [R2];
        /*0078*/                   ISCADD R6.CC, R0, c[0x0][0x150], 0x2;
        /*0088*/                   IMAD.HI.X R7, R0, R7, c[0x0][0x154];
        /*0090*/                   FADD R0, R2, R4;
        /*0098*/                   ST.E [R6], R0;
        /*00a0*/                   EXIT;
.L_1:
        /*00a8*/                   BRA `(.L_1);
.L_24:

// Legend:
//     ^        : Register assignment
//     v        : Register usage
//     x        : Register usage and reassignment
//     :        : Register in use
//     <space>  : Register not in use
//     #        : Number of occupied registers
```

.Fig. 1. Vector Add Kernel Register Liveness

Analysis Control-Flow-Graph:vectorAdd Kernel Benchmark

In the proposed work we performed the analysis on AlexNet benchmark of the Tango Benchmark Suite.

B. Register Liveness Analysis

Data Generation/Analysis Flow:

- Generation of Control-Flow-Graph for the SASS code (nvdisasm tool and Graphviz)
- Performing register Liveness Analysis on the generated CFG (Nsight tool)
- Plotting the graph between the actual required registers versus statically allocated registers over the span of the kernel (percentage of static allocated registers vs. required live registers for each instruction)

VI. OBSERVATION AND ANALYSIS

Analysis is done on the basis of the observations obtained from the following three graphs:

A. Live Register Count vs. percentage of Static Instructions.
B. Live Register Count vs. percentage of Warp Execution.
C. Live Register Count vs. percentage of Dynamic instructions.

Graph A provides an insight into how many instructions of the src assembly code can be executed for a given number of registers. Graph B demonstrates the register count for the registers allocated and used by the warp during execution. Graph C provides an insight into how many instructions really executed for a given number of registers[29].

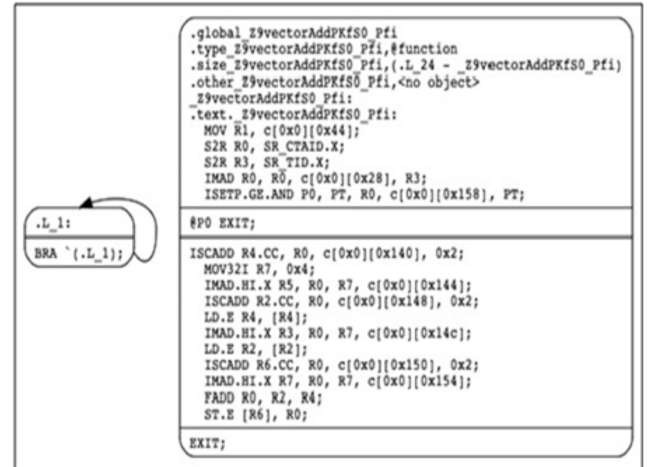


Fig. 2. Control Flow Graph

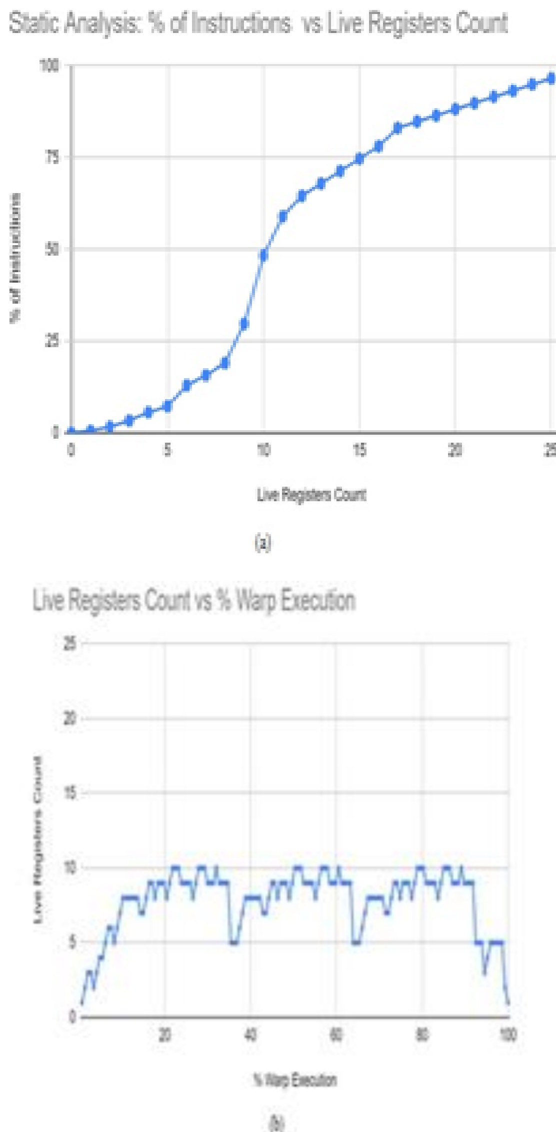
(Vector Add Kernel)

A. Kernel 1: executePoolingCuda (Fig. 3)

In this kernel, there is a scope for early release of the registers since for the last 5.5% of the warp execution, maximum 5 registers are required. The max register requirement computed at the time of compilation is 25. Static analysis shows that 75% of the instructions need 15 registers to be live at any instant; hence our hybrid approach allocates 15 registers from the static pool. Any extra register requirements would be allocated from the dynamic pool at the execution time. However, the dynamic analysis performed by executing the kernel shows that all the instructions got executed with the 10 registers allocated from the static pool, so no further request was needed to the dynamic register pool. Another aspect to observe

Engineering and Technology Journal for Research and Innovation (ETJRI)
ISSN 2581-8678, Volume IV, Issue II, Jun 2022

is that early release technique[1] would require 25 registers for 94.55% of the warp execution and 5 registers for 5.45% at the tail end, which on average requires 24 registers across the complete warp execution. Our hybrid approach, whereas allocates 15 registers for the complete warp execution, if combined with early release approach, then on average it would need 14.55 registers(i.e. 15 registers). Compared to the default GPU allocation scheme, hybrid approach is using 37.5% (15 instead of 25) less registers for this particular kernel[30].



Dynamic Analysis: % of Instructions vs Live Registers Count

% of total Instructions

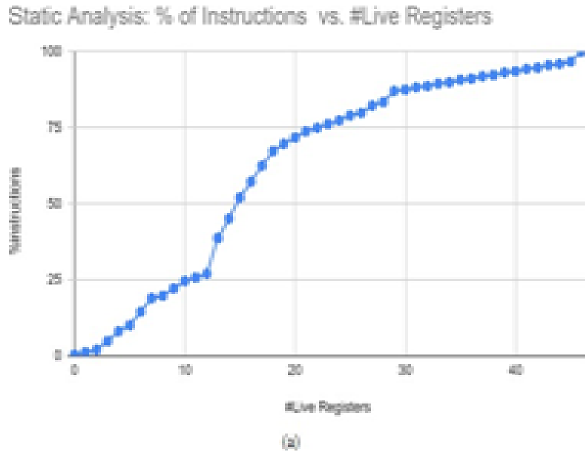
Live Registers Count

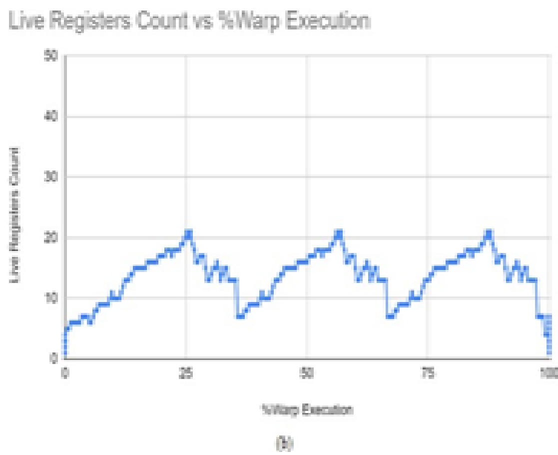
(c)

Fig. 3. ExecutePoolingCuda

B. Kernel 2: Executing 3D Convolution Group2Cuda (Fig.4) and execute 3D Convolution (Fig. 5)

In execute3DConvolutionGroup2Cuda kernel, there is no scope for early release of the registers since it releases the registers only for 0.03% of the warp execution at the tail end. The max register requirement computed at the time of compilation is 46. Static analysis shows that 75% of the instructions need 22 registers to be live at any instant; hence our hybrid approach allocates 23 registers from the static pool. Any extra register requirement would be allocated from the dynamic pool at the execution time. However, the dynamic analysis performed by executing the kernel shows that all the instructions got executed with the 21 registers allocated from the static pool, so no further request was needed to the dynamic register pool. execute3DConvolution shows similar behaviour.





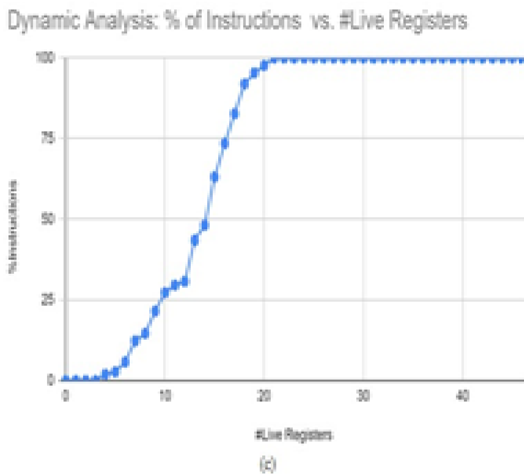


Fig. 4. Execute 3D Convolution Group2 Cuda

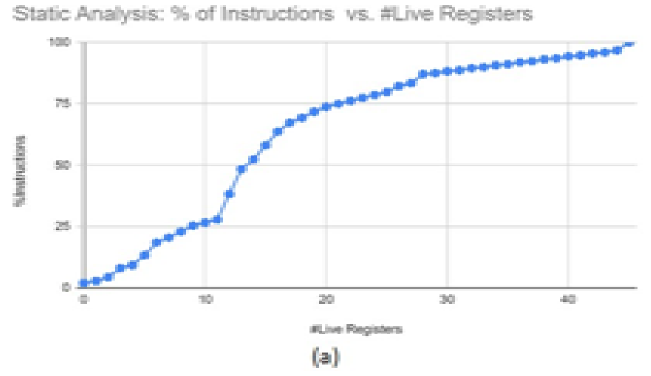
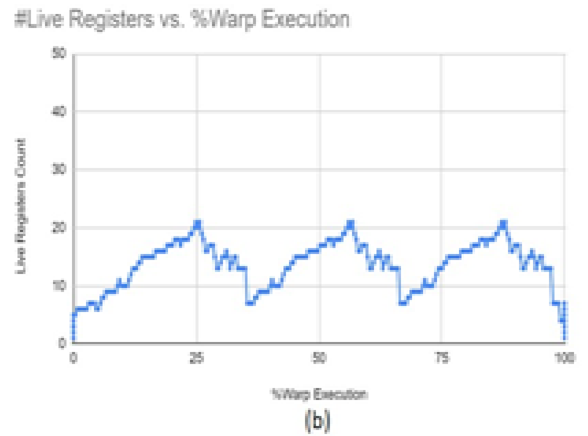


Fig. 5. Execute 3D Convolution Cuda



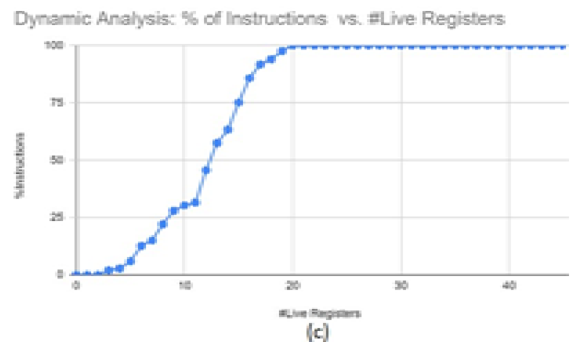
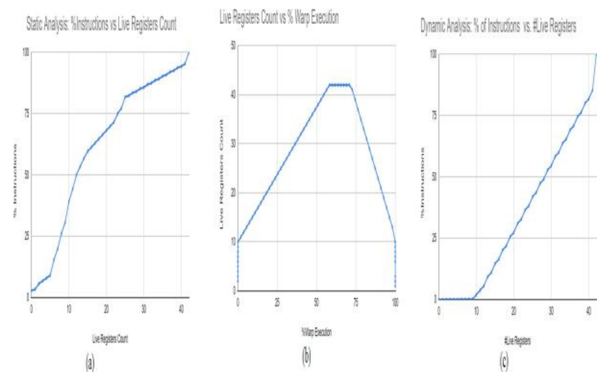


Fig. 6. executeFCLayer



Engineering and Technology Journal for Research and Innovation (ETJRI)
ISSN 2581-8678, Volume IV, Issue II, Jun 2022

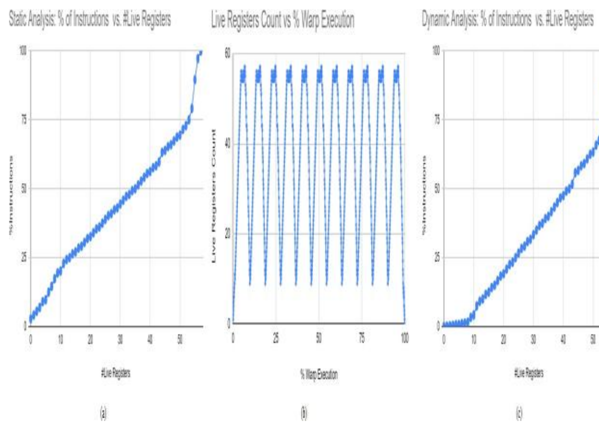


Fig. 7. firstLayer

	Max Register Requirement based on Compile time	3/4th of instructions can be executed with register	Default Policy Register Allocation Count	Hybrid Approach Allocation Count (Static Pool)	Percentage of instructions satisfied with static pool	Early Release (% warp execution remaining)/ Registers Released	Early Release + Default Policy (Weighted Average)	Early Release + Hybrid Policy (Weighted Average)	Dynamic Pool Register Request Count
executePoolingCuda	25	15	25	15	100%	5.45% / 6	24.05	14.55	0
execute3DConvolutionGroup2Cuda	46	22	46	23	100%	0.03% / 7	46	23	0
execute3DConvolution	45	21	45	23	100%	0.01% / 14	45	23	0
executeFCLayer	42	23	42	23	36.5%	0.05% / 10	42	23	19
firstLayer	57	53	57	53	70%	1% / 9	56.52	52.56	4

Fig. 8. Comparison of Register Allocation Schemes for various AlexNet Kernels

C. Kernel 3: execute FC Layer (Fig. 6)

In execute FC Layer kernel, there is no scope for early release of the registers since it releases the registers only for 0.05% of the warp execution at the tail end. The max register requirement computed at the time of compilation is 42. Static analysis shows that 75% of the instructions need 23 registers to be live at any instant; hence our hybrid approach allocates 23 registers from the static pool. Any extra register requirements would be allocated from the dynamic pool at the execution time. However, the dynamic analysis performed by executing the kernel shows that the instructions need all 42 registers at some point of warp execution, resulting in the request to be made to dynamic pool for the remaining 19 registers in steps based on the Basic Block requirements. This could result in some overhead.

D. Kernel 4: first Layer (Fig. 7)

In firstLayer kernel, there is no scope for early release of the registers since it releases the registers only for 1% of the warp execution at the tail end. The max register requirement computed at the time of compilation is 57. Static analysis shows that 75% of the instructions need 53 registers to be live at any instant; hence our hybrid approach allocates 53 registers from the static pool. Very less requirement needs to be fulfilled from the dynamic pool at the execution time. However, the dynamic analysis performed by executing the kernel shows that only 70% of the instructions execute with 53 registers. So here hybrid approach does not perform well. Number of the registers allocated from the static pool is given by the maximum(A,B) where A is half of the Maximum register requirement based on the compile time and B is the number of register required for 75% of instructions to get executed.

VII. CONCLUSION

There is a high need to move from the static and exclusive allocation of the physical registers in order to increase the thread occupancy to increase the GPU throughput. From the analysis of the five kernels of the AlexNet benchmark, we observed that our hybrid approach could

be a better alternative to the default static register allocation scheme. To gain better insight on the percentage of utilisation gain obtained from hybrid approach in general, we need to perform our analysis on various benchmark suites. In future, we can implement our approach on GPGPU- Sim to analyze the complete solution and related issues as well. Deadlock is one of the issues our approach needs to figure out as two warps requesting the registers from the dynamic pool should not end up in deadlock. One needs to keep in mind that the mechanism should not add a huge hardware overhead in terms of space and complexity or else the solution would not be too much feasible to see the light.

REFERENCES

[1] D. Voitsechov, A. Zulfiqar, M. Stephenson, M. Gebhart, and S. W. Keckler, “Software-directed techniques for improved GPU register file utilization,” ACM Trans. Archit. Code Optim., 2018.

[2] P. Sakdhnagool, A. Sabne, and R. Eigenmann, “RegDem: Increasing GPU performance via shared memory register spilling,” arXiv. 2019.

[3] Y. P. You and S. C. Chen, “VecRA: A vector-aware register allocator for GPU shader processors,” in ACM Transactions on Embedded Computing Systems, 2016.

[4] S. Cook, CUDA Programming: A Developer’s Guide to Parallel Computing with GPUs. 2013.

[5] H. Asghari Esfeden, F. Khorasani, H. Jeon, D. Wong, and N. Abu-Ghazaleh, “CORF: Coalescing Operand Register File for GPUs,” in International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS, 2019.

[6] K. Iyer and J. Kiel, “GPU debugging and profiling with NVIDIA parallel nsight,” in Game Development Tools, 2016.

[7] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan, “Improving GPGPU concurrency with elastic kernels,” ACM SIGARCH Comput. Archit. News, 2013.

[8] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo, “Simultaneous Multikernel GPU: Multi-tasking throughput processors via fine-grained sharing,” in Proceedings - International Symposium on High-Performance Computer Architecture, 2016.

[9] C. Gregg, J. Dorn, K. Hazelwood, and K. Skadron, “Fine-grained resource sharing for concurrent GPGPU kernels,” in 4th USENIX Workshop on Hot Topics in Parallelism, HotPar 2012, 2012.

[10] J. T. Adriaens, K. Compton, N. S. Kim, and M. J. Schulte, “The case for GPGPU spatial multitasking,” in Proceedings - International Symposium on High-Performance Computer Architecture, 2012.

[11] M. Lee et al., “Improving GPGPU resource utilization through alternative thread block scheduling,” in Proceedings - International Symposium on High-Performance Computer Architecture, 2014.

[12] J. Liu, J. Yang, and R. Melhem, “SAWS: Synchronization aware GPGPU warp scheduling for multiple independent warp schedulers,” in Proceedings of the Annual International Symposium on Microarchitecture, MICRO, 2015.

[13] Y. Liu et al., “Barrier-aware warp scheduling for throughput processors,” in Proceedings of the International Conference on Supercomputing, 2016.

[14] S. Y. Lee and C. J. Wu, “CAWS: Criticality-aware warp scheduling for GPGPU workloads,” in Parallel Architectures and Compilation Techniques - Conference Proceedings, PACT, 2014.

[15] S. Y. Lee, A. Arunkumar, and C. J. Wu, “CAWA: Coordinated warp scheduling and cache prioritization for critical warp acceleration of GPGPU workloads,” in Proceedings - International Symposium on Computer Architecture, 2015.

[16] P. Xiang, Y. Yang, and H. Zhou, “Warp-level divergence in GPUs: Characterization, impact, and mitigation,” in Proceedings - International Symposium on High-Performance Computer Architecture, 2014.

[17] N. Vijaykumar et al., “Zorua: A holistic approach to resource virtualization in GPUs,” in Proceedings of the Annual International Symposium on Microarchitecture, MICRO, 2016.

[18] M. Gebhart, S. W. Keckler, B. Khailany, R. Krashinsky, and W. J. Dally, “Unifying primary cache, scratch, and register file memories in a throughput processor,” in Proceedings - 2012 IEEE/ACM 45th

International Symposium on Microarchitecture, MICRO 2012, 2012.
[19] Z. Lin, L. Nyland, and H. Zhou, “Enabling Efficient Preemption for SIMT Architectures with Lightweight Context Switching,” in International Conference for High Performance Computing, Networking, Storage and Analysis, SC, 2016.
[20] A. Anand, A. Raj, R. Kohli, and V. Bibhu, “Proposed symmetric key cryptography algorithm for data security,” in 2016 1st International Conference on Innovation and Challenges in Cyber Security, ICICCS 2016, 2016.
[21] S. Chowdhury and P. Mayilvahanan, “A survey on internet of things: Privacy with security of sensors and wearable network ip/protocols,” Int. J. Eng. Technol., 2018.
[22] S. Dublish, V. Nagarajan, and N. Topham, “Characterizing memory bottlenecks in GPGPU workloads,” in Proceedings of the 2016 IEEE International Symposium on Workload Characterization, IISWC 2016, 2016.
[23] S. Verma, S. Kaur, G. Dhiman, and A. Kaur, “Design of a novel energy efficient routing framework for Wireless Nanosensor Networks,” in ICSCCC 2018 - 1st International Conference on Secure Cyber Computing and Communications, 2018.
[24] G. Dhiman, “ESA: a hybrid bio-inspired metaheuristic optimization approach for engineering problems,” Eng. Comput., 2019.
[25] N. Kumar, N. Kharkwal, R. Kohli, and S. Choudhary, “Ethical aspects and future of artificial intelligence,” in 2016 1st International Conference on Innovation and Challenges in Cyber Security, ICICCS 2016, 2016.
[26] P. Singh, G. Dhiman, and A. Kaur, “A quantum approach for time series data based on graph and Schrödinger equations methods,” Mod. Phys. Lett. A, 2018.
[27] K. B. Prakash, S. Nazeer, P. K. Vadla, and S. Chowdhury, “Layered programming model for resource provisioning in fog computing using yet another fog simulator,” Int. J. Emerg. Trends Eng. Res., 2020.
[28] R. Nair, S. Gupta, M. Soni, P. Kumar Shukla, and G. Dhiman, “An approach to minimize the energy consumption during blockchain transaction,” Mater. Today Proc., 2020.
[29] G. Dhiman and V. Kumar, “Multi-objective spotted hyena optimizer: A Multi-objective optimization algorithm for engineering problems,” Knowledge-Based Syst., 2018.
[30] G. Dhiman, “MOSHEPO: a hybrid multi-objective approach to solve economic load dispatch and micro grid problems,” Appl. Intell., 2020.

Engineering and Technology Journal for Research and Innovation (ETJRI)
ISSN 2581-8678, Volume IV, Issue II, Jun 2022

Engineering and Technology Journal for Research and Innovation (ETJRI)
ISSN 2581-8678, Volume V, Issue I, Dec 2022

Engineering and Technology Journal for Research and Innovation (ETJRI)
ISSN 2581-8678, Volume IV, Issue II, Jun 2022