Throughput Optimization and Resource Allocation on GPUs under Multi-Application Execution

Srinivasa Reddy Punyala, Theodoros Marinakis, Arash Komaee, Iraklis Anagnostopoulos
Department of Electrical and Computer Engineering, Southern Illinois University, Carbondale, U.S.A.

***Abstract*—Platform heterogeneity prevails as a solution to the throughput and computational challenges imposed by parallel applications and technology scaling. Specifically, Graphics Processing Units (GPUs) are based on the Single Instruction Multiple Thread (SIMT) paradigm and they can offer tremendous speed-up for parallel applications. However, GPUs were designed to execute a single application at a time. In case of simultaneous multi-application execution, due to the GPUs' massive multi-threading paradigm, applications compete against each other using destructively the shared resources (caches and memory controllers) resulting in significant throughput degradation. In this paper, a methodology for minimizing interference in shared resources and provide efficient concurrent execution of multiple applications on GPUs is presented. Particularly, the proposed methodology (i) performs application classification; (ii) analyzes the per-class interference; (iii) finds the best matching between classes; and (iv) employs an efficient resource allocation. Experimental results showed that the proposed approach increases the throughput of the system for two concurrent applications by an average of 36% compared to the default execution and 10% compared to an exahustive profile-based optimization technique.**

I. Introduction and Motivation

Modern computing and embedded systems are characterized by the increased number of integrated processing elements and follow the many-core architecture paradigm leaving behind conventional superscalar computing architectures. This evolution and increase in the number of employed processors, has also driven rapid changes on the software side and specifically to multi-threaded applications which have become more dynamic, memory- and compute-intensive. Modern systems that rely completely on CPUs cannot meet the requirements of memory intensive applications and particularly for applications that rely on big data sets.

Platform heterogeneity prevails as a solution to the throughput and computational challenges imposed by parallel applications and technology scaling. Specifically, Graphics Processing Units (GPUs) are becoming an inevitable part of heterogeneous systems due to massive multi-threading and fast context switching [1]. Originally, GPUs were designed to accelerate graphics rendering applications [2] as they could handle more efficiently the required workload and operations. However, as the technology on GPU resources and the software support is improved [3], GPUs now have the ability to process massive number of threads concurrently establishing the term General Purpose GPUs (GPGPUs) and making them cornerstone in modern data centers. Currently in Top500 [4], more than 60% of the accelerators are GPUs.

GPUs are based on the Single Instruction Multiple Thread (SIMT) paradigm and can offer tremendous speed-up for massively parallel applications that exploit the thread and data level parallelism. The characteristic of GPU applications is that they can tolerate long latencies, due to their inherent latency hiding ability [5], but they are more bandwidth sensitive.

Originally, GPUs were designed to execute a single application at a time and in order to further improve its performance GPUs employed different levels of cache and multiple memory controllers. The main reason for that lays in GPU architecture design. A GPU consists of multiple cores, also called as *Streaming Multiprocessors (SMs)*, and in each SM hardware threads are organized in warps. From the software perspective, a GPU application consists of kernels organized as blocks or *Cooperative Threads Arrays (CTA)*.

Multiple application execution can be performed in a *temporal* or *spatial* way. The temporal approach uses time multiplexing in order to allocate resources to different users and it is the common technique in GPU virtualization [2]. However, this approach leads to system underutilization and poor performance [6]. Unlike CPUs, where multi-application is architecturally supported concurrent execution of multiple applications on GPUs prevails as challenge in order to unlock system's performance [6]–[8]. Due to massive application parallelism and numerous generated threads, GPUs' performance is affected by contention on shared resources in multiple ways. SMs are not independent processing elements but share resources, such as caches and memory controllers resulting in performance degradation. Threads, when running simultaneously, compete against each other using destructively the shared resources [9].

In this paper, we present a methodology for efficient concurrent execution of multiple applications on GPUs by minimizing the interference in shared resources. Specifically, the proposed methodology focuses on the maximization of GPU's throughput by (i) performing application classification; (ii) analyzing the per-class interference and slow-down; (iii) finding the best matching between classes; and (iv)employing an efficient kernel-to-SM policy that reduces the destructive effects of applications' interference.

II. Related Work

Improving performance and throughput of GPUs has been researched previously in the context of thread-level parallelism, concurrent kernel execution and shared resource contention. Authors in [2] proposed to use large warps to improve the performance of GPU applications in case of branch divergence within warps. Even though this technique improves the performance of the application, it cannot improve the device utilization if the application does not exploit thread-level parallelism. In [6], the authors proposed a technique to improve kernel concurrency using elastic kernels. However, their technique cannot be applied to every kernel. Specifically, the presented scheme will not work for kernels that are effected when the hardware thread ids are different from software ids. Additionally, the authors in [10] present a warp-aware memory scheduling method that focuses on minimizing inter-warp contention.

Current generations of GPUs support concurrent execution for kernels of the same application. However, device utilization is still limited by the data dependencies. Unlike this, in current paper *we propose to run multiple applications of selected classes* to achieve higher device utilization. Traditional way for multiple application execution on GPUs is time sharing. Nonetheless, this adds a high overhead of context switching. The authors in [11] proposed a method to assign to GPUs the required bandwidth and reduce contention with other devices. The authors in [12] proposed a technique to run kernels concurrently from different contexts through context funneling. NVIDIA introduced CUDA MPS [13] to support running kernels from multiple contexts. However, the kernels are not actually executed concurrently by the device. The MPS server serves a context only after the previous one has finished. In the scope of this work, *we use automatic context funneling* provided by CUDA API along with *hyper Q mechanism* to run multiple applications concurrently.

Additionally, authors in [14] proposed a mechanism for simultaneous kernel execution. In this work a portion of threads from an already running kernel are preempted and the freed resources are given to a new incoming kernel. This mechanism still involves partial context switching. However, the authors do no consider kernels from different applications on different streams. In [2], [8], a memory scheduling policy for GPUs that supports concurrent application execution is proposed. The scheme improves device throughput by reducing the contention in shared resources. Whereas, in the presented work *we co-schedule applications that have less contention* while improving the utilization of the device. The authors in [15] proposed to execute multiple applications concurrently on a GPU through resource partitioning. However, the selection of the applications is performed in FCFS way which does not improve the throughput of the device in a long run. Last, in [7], [16] GPU resource partitioning policies for multiple application execution on GPUs are presented.

III. Methodology

As aforementioned, a GPU consists of multiple streaming-multiprocessors (SMs) which share an L2 cache and memory controller. Each SM follows the Single Instruction Multiple Thread (SIMT) paradigm, with its own L1 cache, and it utilizes a warp pool. Each warp contains a group of threads to be executed, which size depends on the architecture (typically 32), and a hardware scheduler is responsible to select a ready for execution warp. Applications spawn threads by employing kernels. Each kernel is organized as thread blocks that are mapped on the SMs. With this massive multi-threading paradigm, applications, when running simultaneously, compete against each other using destructively the shared resources and they are slowed down compared to running alone [2], [15].

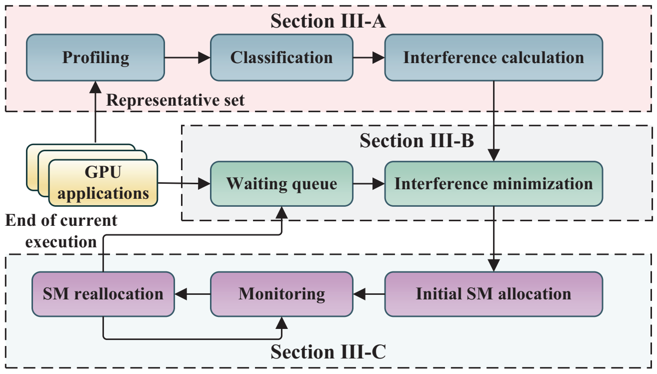


Fig. 1: Overall flow of the proposed methodology

The goal of the proposed methodology is to maximize the throughput of a GPU under the scenario of multi-application execution. As throughput we define $T = \frac{I_T}{C_T}$ where I_T is the total number of instructions, C_T is the total number of cycles. Figure 1 depicts an overview of the proposed methodology consisting of three steps. The first step (Section III-A) includes application profiling, classification and inter-class interference calculation. In application profiling, the goal is to extract information regarding the usage of an application on the shared resources and classify it accordingly. Then, we measure the slow down that happens when applications of two different classes are executed together. Having calculated the inter-class contention, the next step (Section III-B) is to find a systematic way to select which applications in the waiting queue will be executed together in order to minimize contention and consequently increase the throughput of the system. Last (Section III-C), the proposed methodology tries to reallocate SMs efficiently at run-time in order to further enhance performance.

A. Classification

The goal of this task is to measure the effect that an application has on other concurrently executing applications. Instead of exploring the effect of each application separately, which is not scalable, we propose a GPU-oriented application classification scheme and study the inter-class impact. Applications with common characteristics can be grouped together forming a class. For that reason, it is crucial to recognize the parts of the system that benefit or suffer from a specific change in order to adapt the system properly.

The proposed classification separates application into 4 different classes: (i) Class M: Applications that belong to this class show high memory activity. They access memory regularly and they bring data to SMs at an increased rate. The primary contention point is the memory bus; (ii) Class MC: Applications that belong to this class have increased memory

TABLE I: Classfication

Class	Classification criteria	Benchamrk
M	$MB > \alpha$	BLK, GUPS
MC	$\beta < MB < \alpha$	BP, FFT, 3DS, LPS, RAY
C	L2→L1 $> \gamma$ $R > \delta$ $IPC < \epsilon$	BFS2, SPMV
A	$R < \delta$ $IPC > \epsilon$	LUD, HS, SAD, NN

activity, which is lower than Class M, but they present higher L2 activity. The contention points are both the memory bus and the L2 at a lower rate. (iii) Class C: Applications of this class have high activity and perform heavy reuse on the cached data of the shared L2; and (iv) Class A: This class consists of computing intensive applications. The impact they have on main memory and L2 is negligible compared to other classes. The actual classification is based on the the data path from main memory to SMs. Specifically, we are focusing on the stream flow across SMs capturing the memory and cache link activity. The metrics we are using are: (i) Memory bandwidth (MB); (ii) L2→L1 bandwidth (L2→L1); (iii) Memory-to-Compute ratio (R); and (iv) Instructions per Cycle (IPC). The information required for these metrics can be easily collected at run-time. Table I presents the classification criteria and the categorization of the Rodinia [17] benchmarks. For the current experimental set up[1], we set the five thresholds that guide the classification as $\alpha = \gamma = 0.55 \times MB_{max}$, $\beta = 0.30 \times MB_{max}$, $\delta = 0.2$ and $\epsilon = 0.20 \times IPC_{max}$. These threshold values are chosen based on the observations we made during offline profiling of the applications.

The step after the applications' classification is to examine how they interact with each other. Testing the different combinations, we can determine the level of contention generated by the co-execution of applications belonging to different classes and at what extent each class is impacted by this contention. We denote as $S_{j \to i}$ the performance slow-down that an application of class i suffers when it is executed together with an application that belongs to class j. For a representative set of applications, we measured the performance slow down of concurrent application execution considering equal distribution of SMs. From the heatmap presented in Figure 2, we come to the following conclusions. The A Class is the one that gets less impacted by the co-execution with the others, experiencing slowdowns ranging from 1.06 to 1.20 (last column). In addition, it causes the least suffering when executed with the others (slowdowns 1.06-1.51). Concerning C applications, they suffer from a moderate slowdown (1.50-1.61) and cause the greatest interference in M Class (x1.76). MC applications are more negatively impacted by the M class (x1.78), while interestingly cause a significant degradation to class M at the extent of 2.05 times. However, the most worth noticing part is the behavior of M applications. Looking at the respective column we realize that they are the most sensitive, as they are affected at the greatest extent by the other classes. In addition to that, they cause the greatest destructive interference to the

[1]Section IV provides our experimental set up

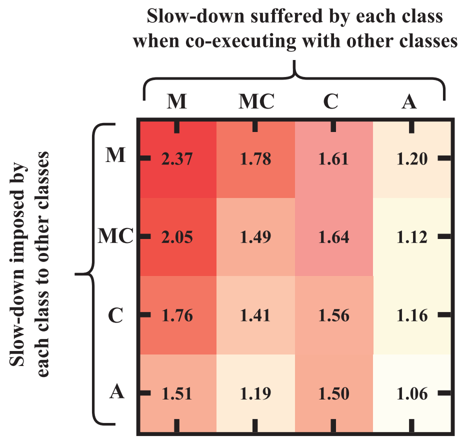


Fig. 2: Average application slow-down due to co-execution

other classes, taking into account the measurements in the respective row. This is explained by the fact that they utilize a great percentage of the available memory bandwidth and any pressure in this resource causes them great performance loss. This characteristic is also the main reason of degradation on the other classes. High memory bandwidth means high L2 trashing and limited bandwidth for the other applications.

B. Minimization of inter-class contention

After having classified and calculated the inter-class slowdown, the next step is to find which applications should be executed together in order to maximize throughput. We present an analytical model that, for a given classification, it finds the best application matching that minimizes the inter-class contention. The proposed methodology works in the granularity of classes reducing the complexity of the solution.

We assume a GPU with N_{SM} number of SMs and we split them into equal N_G groups. Each group N_i has the same number of SMs ($\frac{N_{SM}}{N_G}$) and we assign one application to one group of SMs for concurrent execution. After classification we have a collection of classes C, N_C is the size of C which gives the total number of classes available. With a queue of applications of N_C classes, the assignment of applications to N_G groups of SMs can be done in

$$N_P = \binom{N_C + N_G - 1}{N_G} \tag{1}$$

ways. Each way of this assignment is called a *pattern*.

For every pattern, we define a vector P_k, with size N_C and $k = 1, 2, \cdots, N_P$, that describes the distribution of classes inside that pattern. As depicted in Equation 2, the i element of the vector corresponds to a number that indicates how many c_i classes exist in the pattern. For example, the element p_1 represents how many c_1 classes exist in the pattern, p_2 for c_2

class etc. Consequently, $\sum_{i=1}^{N_C} P_k[i] = N_G$.

$$P_k = \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ \vdots \\ p_{N_C} \end{bmatrix} = \begin{bmatrix} \#c_1 = \{0, 1, \cdots, N_G\} \\ \#c_2 = \{0, 1, \cdots, N_G\} \\ \#c_3 = \{0, 1, \cdots, N_G\} \\ \cdots \\ \#c_{N_C} = \{0, 1, \cdots, N_G\} \end{bmatrix} \tag{2}$$

Last, we define the efficiency e_k of each pattern P_k in Equation 3.

$$e_k = \frac{1}{N_G}\Big(\sum_{\forall i,j \in P_k}^{i \neq j} \frac{1}{S_{i \to j}} \Big) \tag{3}$$

where $k = 1, 2, \cdots, N_P$.

We suppose that our system has a queue, with size N_q, where incoming applications are stored. Each application belongs to a specific class, but the actual distribution of the classes is considered random. Thus, $N_q = N_q^1 + N_q^2 + \cdots + N_q^{N_C}$, where N_q^i refers to the number of applications inside the queue that belong to the class i. Assuming, without loss of generality, that $\frac{N_q}{N_G} = L$ is always an integer. To obtain a maximum throughput we obtain a function f shown in Equation 4, in terms of efficiencies of all the patterns.

$$f = e_1 L_1 + e_2 L_2 + \cdots + e_{N_P} L_{N_P} \tag{4}$$

We then *find* $L_1, L_2, \cdots, L_{N_P}$ that maximize the value of f. The problem can be formulated as an Integer Linear Programming (ILP) problem described by Equation 5.

$$\begin{aligned} & max\{e_1 L_1 + e_2 L_2 + \cdots + e_{N_P} L_{N_P}\} \\ & \text{subject to} \\ & L_1 + L_2 + \cdots + L_{N_P} = L \\ & \begin{bmatrix} P_1 & P_2 & \cdots & P_{N_P} \end{bmatrix} \begin{bmatrix} L_1 \\ L_2 \\ \vdots \\ L_{N_P} \end{bmatrix} = \begin{bmatrix} N_q^1 \\ N_q^2 \\ \vdots \\ N_q^{N_C} \end{bmatrix} \end{aligned} \tag{5}$$

It is worth mentioning, that the complexity of Equation 5 depends mostly on the number of classes N_C. The optimal solution is to consider that each application is a separate class. In this way, Equation 5 gives the best matching but requires exhaustive offline exploration and as the size of the queue increases the number of patterns becomes unmanageable. Thus, keeping the number of classes small but representative enough, and having extracted the inter-class slow-down, the only thing that is needed is the class of the application.

C. SM Reallocation

By solving Equation 5, we can find which applications, based on their classes, should be executed together in order to minimize interference. This step focuses on run-time SM reallocation in order to further boost performance.

The algorithm for reallocation of SMs, under multi-application execution, is presented in Algorithm 1. Initially all applications start with equal number of allocated SMs and a score value of 0. The algorithm after every T_C cycles checks the device throughput and performance statistics for each executing application.

Based on the values, each application changes its score. If the Instructions Per Cycle (IPC) of an application App_i is less than a value IPC_{thr}, the score of this application is $V[i] = 1$. If the bandwidth utilization is greater than a value BW_{thr} then $V[i] = 2$ and if both conditions are true then $V[i] = 3$. A high score means that the application negatively affects the throughput of the device. This happens because an application with low IPC and high memory bandwidth relies on data transfer and some of the SMs that have been allocated can be used by another compute intensive application increasing the throughput of the GPU in total. Then, based on the score of each application, we deallocate n_r SMs from the application with the highest score and we allocate them to the application with the lowest score. When the allocated resources for an application reaches R_{min}, the realocation stops for the specific pair.

The SM deallocation can be done in three ways. The first method requires partial context switching [14] which is expensive in terms of latency and interconnect bandwidth. The second way, is to completely discard the running kernel on the selected SMs. However, this approach imposes big performance slow-down. The last way is to let the selected SMs finish the currently running blocks and once they are finished, they are transfered to the other application. Algorithm 1 follows the third method, even though has a small performance overhead, allows for smooth exchange of SMs at run-time.

IV. Experimental Results

In order to validate our methodology we have performed extensive simulation experiments using GPGPU-sim [18], a cycle-level accurate simulator for GPUs, that supports NVIDIA CUDA, and Rodinia [17] benchmarks as high performance parallel applications. GPGPU-sim was modified in order to support multiple streams and concurrent application execution. The experimental set up is described in Table II.

TABLE II: Experimental set up

GPU Architecture	
# of SMs	60
Core frequency	700MHz
Warps per SM	48
Blocks per SM	8
Shared Memory	48kB
L1 Data cache	16kB per SM
L1 Instr. cache	2kB per SM
L2 cache	768kB
Warp scheduler	GTO [19]

Regarding the simultaneous application execution we created a queue with multiple Rodinia benchmarks and two cases were tested: (i) 2 concurrent applications on GPU (queue size of 20 applications); and (ii) 3 concurrent applications on GPU (queue size of 36 applications). In both cases we evaluated the following scenarios regarding the distribution of the applications to be executed: Equal distribution: The

Algorithm 1 SM Re-allocation algorithm

```
$T$:Current Throughput
$Tp$:Previous Throughput
$N$:Total number of SMs
$n$:Number of applications running concurrently
$R_i$:Number of SMs for $i_{th}$ application
$V[i]$:Score of $i_{th}$ application
$R_{min}$:Minimum SMs Required for an application
BWutil:Memory bandwidth utilization
$n_r$:number of SMs to transfer

1: Initial:
2: for each app do
3:     Intialize score of each app to 0;
4:     allocate equal resources to each app;
5: end for
6:
7: for every $T_C$ cycles do
8:     while $T > T_p$ and $R_i > R_{min}$ do
9:         for each app do
10:            if $IPC[i] < IPC_{thr}$ then
11:                Increment score v[i] by 1;
12:            end if
13:            if $BWutil[i] > BW_{thr}$ then
14:                Increment score v[i] by 2;
15:            end if
16:        end for
17:        if all applications have same score then
18:            break;
19:        else
20:            Free $n_r$ SMs from app with high score;
21:            Give the freed SMs to app with less score;
22:        end if
23:    end while
24:    if $T < T_P$ then
25:        go back to previous allocation
26:    end if
27:    V[i]=0
28: end for
```

distribution of classes (Section III-A) inside the queue was the same for all. M-oriented workload: In this scenario, the Class M dominates in queue with more than 50% of the applications. The distribution of the other classes remains equal. MC-oriented workload: In this scenario, the Class MC dominates in queue with more than 50% of the applications. The distribution of the other classes remains equal. C-oriented workload: In this scenario, the Class C dominates in queue with more than 50% of the applications. The distribution of the other classes remains equal. A-oriented workload: In this scenario, the Class A dominates in queue with more than 50% of the applications. The distribution of the other classes remains equal.

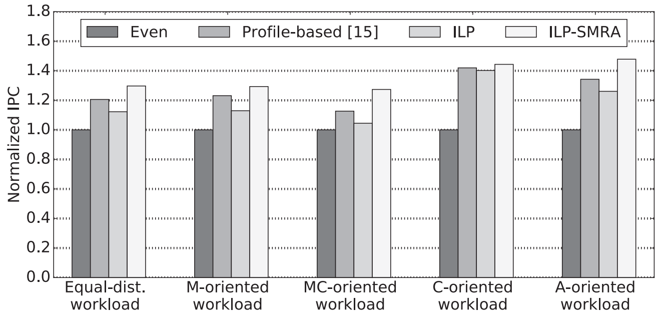
We name the methodology presented in Section III-B as *ILP* and its combination with the SM reallocation technique as *ILP-SMRA*. We compare both methodologies, in the two aforementioned cases (2 and 3 concurrent applications), to: (i) An *Even* approach that distributes SMs among applications evenly and selects the pairs for co-execution based on the arrival order. For example, the first pair/three applications in queue are executed together, then the next pair/three etc. (ii) The *Profile-based* method presented in [15]. This approach, requires extensive application profiling in isolation before the execution in order to identify the number of required SMs that maximizes their performance. The execution of the applications is performed based on the arrival order and the SM distribution is based on the off-line extracted number.

Figure 3a presents the normalized throughput of the GPU in the case of two concurrent application execution under different workload distributions. The *Even* approach is considered as the baseline for our comparison. The *ILP* method increased throughput by an average of 19% achieving the best gain of 40% in the MC-oriented workload. As aforementioned, the *ILP* method focuses on finding the best application matching in order to reduce contention working at the granularity of classes. *ILP-SMRA* increases throughput by an average of 36%, compared to the *Even* method, achieving the best gain of 48% in the A-oriented workload. *ILP-SMRA* not only reduces contention due to the best matching of the classes, but it performs run-time SM reallocation that further boost the performance of the GPU. As expected, the *Profile-based* method [15] achieved on average 26% better throughput compared to the *Even* method and it was better than *ILP* by 7% on average and worse by *ILP-SMRA* by 10% on average. However, it requires extensive off-line profiling in order to find the best configuration for each application but it does not take into consideration the inter-class contention. Figure 3b presents the normalized throughput of the GPU in the case of three concurrent application execution under different workload distributions. The *Even* approach is considered as the baseline for our comparison. *ILP-SMRA* increases throughput by an average of 23%, compared to the *Even* method, achieving the best gain of 40% in the A-oriented workload. Even in the case of three simultaneous executing applications, *ILP-SMRA* reduces contention due to the best matching of the classes and reallocates SMs at run-time based on the score of each application further increasing the throughput of the GPU. Regarding the *Profile-based* method [15], it achieves on average 23% better throughput compared to the *Even* and performs similarly with the *ILP-SMRA*. However, as aforementioned, it requires extensive off-line profiling in order to find the best configuration which does not make it scalable and adaptive to new incoming applications.

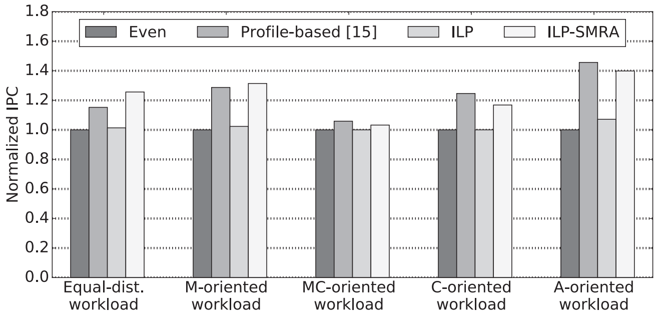
Figure 4 presents the normalized comparison of average IPC (Instructions Per Cycle) per application for the tested use cases having *Even* as the baseline. As depicted in Figure 4a, for 2 concurrent applications, *ILP* achieved on average a gain of 23% while *ILP-SMRA* increased the average IPC gain by 44% on average. Figure 4b depicts the comparison for 3 concurrent applications. And in this case, *ILP* achieved on average a gain of 28% while *ILP-SMRA* increased the average IPC gain by 67% on average.

V. Conclusion

In this paper, a methodology for efficient concurrent execution of multiple applications on GPUs by minimizing the interference in shared resources was presented. Specifically, the proposed methodology focuses on the maximization of GPU's throughput by (i) performing application classification; (ii) analyzing the per-class interference and slow-down; (iii) finding the best matching between classes; and (iv) it employs an efficient kernel-to-SM policy that reduces the destructive effects of applications' interference. Experimental results

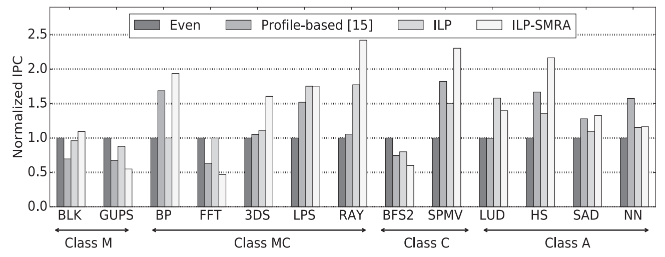


(a) Concurrent execution of 2 applications

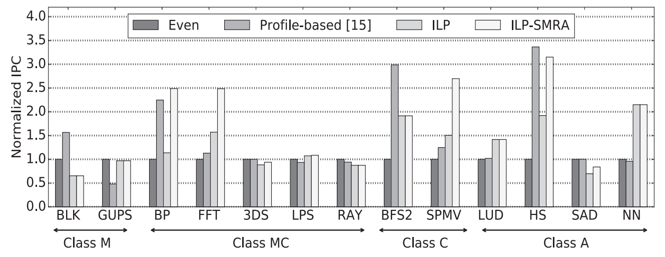


(b) Concurrent execution of 3 applications

Fig. 3: GPU throughput comparison of simultaneous executing applications under different workload distribution



(a) Concurrent execution of 2 applications



(b) Concurrent execution of 3 applications

Fig. 4: Normalized comparison of average IPC per application in concurrent execution

showed that the proposed approach increases the throughput of the system for two concurrent applications by an average of 36% compared to the default execution and 10% compared to an exahustive profile-based optimization technique [15].

REFERENCES

[1] O. Kayiran, N. C. Nachiappan, A. Jog, R. Ausavarungnirun, M. T. Kandemir, G. H. Loh, O. Mutlu, and C. R. Das, "Managing gpu concurrency in heterogeneous architectures," in *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*. IEEE, 2014.

[2] A. Jog, O. Kayiran, T. Kesten, A. Pattnaik, E. Bolotin, N. Chatterjee, S. W. Keckler, M. T. Kandemir, and C. R. Das, "Anatomy of gpu memory system for multi-application execution," in *Proceedings of the 2015 International Symposium on Memory Systems*. ACM, 2015.

[3] S. R. Agrawal, V. Pistol, J. Pang, J. Tran, D. Tarjan, and A. R. Lebeck, "Rhythm: Harnessing data parallel hardware for server workloads," in *ACM SIGARCH Computer Architecture News*, vol. 42, no. 1. ACM, 2014, pp. 19–34.

[4] Top500, "Top500 Supercomputers, www.top500.org."

[5] S.-Y. Lee, A. Arunkumar, and C.-J. Wu, "Cawa: coordinated warp scheduling and cache prioritization for critical warp acceleration of gpgpu workloads," in *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3. ACM, 2015, pp. 515–527.

[6] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan, "Improving gpgpu concurrency with elastic kernels," in *ACM SIGPLAN Notices*, vol. 48, no. 4. ACM, 2013, pp. 407–418.

[7] P. Aguilera, K. Morrow, and N. S. Kim, "Qos-aware dynamic resource allocation for spatial-multitasking gpus," in *Design Automation Conference (ASP-DAC), 2014 19th Asia and South Pacific*. IEEE, 2014.

[8] A. Jog, E. Bolotin, Z. Guz, M. Parker, S. W. Keckler, M. T. Kandemir, and C. R. Das, "Application-aware memory system for fair and efficient execution of concurrent gpgpu applications," in *Proceedings of workshop on general purpose processing using GPUs*. ACM, 2014, p. 1.

[9] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt, "Improving gpu performance via large warps and two-level warp scheduling," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2011.

[10] N. Chatterjee, M. O'Connor, G. H. Loh, N. Jayasena, and R. Balasubramonian, "Managing dram latency divergence in irregular gpgpu applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2014, pp. 128–139.

[11] M. K. Jeong, M. Erez, C. Sudanthi, and N. Paver, "A qos-aware memory controller for dynamically balancing gpu and cpu bandwidth use in an mpsoc," in *Proceedings of the 49th Annual Design Automation Conference*. ACM, 2012, pp. 850–855.

[12] L. Wang, M. Huang, and T. El-Ghazawi, "Exploiting concurrent kernel execution on graphic processing units," in *High performance computing and simulation (HPCS), 2011 international conference on*. IEEE, 2011.

[13] F. Wende, T. Steinke, and F. Cordes, "Multi-threaded kernel offloading to gpgpu using hyper-q on kepler architecture," *ZIB-Rep. 14-19 June 2014*, 2014.

[14] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo, "Simultaneous multikernel: Fine-grained sharing of gpus," *IEEE Computer Architecture Letters*, vol. 15, no. 2, pp. 113–116, 2016.

[15] J. T. Adriaens, K. Compton, N. S. Kim, and M. J. Schulte, "The case for gpgpu spatial multitasking," in *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*. IEEE, 2012.

[16] P. Aguilera, K. Morrow, and N. S. Kim, "Fair share: Allocation of gpu resources for both performance and fairness," in *Computer Design (ICCD), 2014 32nd IEEE International Conference on*. IEEE, 2014, pp. 440–447.

[17] 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 *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. Ieee, 2009, pp. 44–54.

[18] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 163–174.

[19] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-conscious wavefront scheduling," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2012, pp. 72–83.