Efficient GPGPU Computing with Cross-Core Resource Sharing and Core Reconfiguration

Ashutosh Dhar
Department of Electrical and Computer Engineering
University of Illinois, Urbana-Champaign
adhar2@illinois.edu

Deming Chen
Department of Electrical and Computer Engineering
University of Illinois, Urbana-Champaign
dchen@illinois.edu

***Abstract*—GPUs are capable of running a variety of applications, however their generic parallel-architecture can lead to inefficient use of resources and reduced power efficiency, due to algorithmic or architectural constraints. In this work, taking inspiration from CGRAs (coarse-grained reconfigurable architectures), we demonstrate resource sharing and re-distribution as a solution that can be leveraged by reconfiguring the GPU on a kernel-by-kernel basis. We explore four different schemes that trade the number of active SMs (streaming multiprocessor) for increased occupancy and local memory resources per SM and demonstrate improved power and energy with limited impact to performance. Our most aggressive scheme, BigSM, is capable of saving energy by up to 54%, and 26% on an average.**

***Keywords*-GPGPU; CGRA; Reconfigurable Architecture**

I. Introduction

The use of GPUs for general purpose computing (GPGPU) has emerged as a dominant paradigm that allows programmers to exploit massively parallel architectures, with relatively simple programming models. The pervasiveness of GPGPU is, in part, due to the generic parallel architecture of the GPU which allows it to run applications from a variety of domains. However, this *one size fits all* architecture makes efficient utilization of resources difficult and can result in lost computational power and reduced energy efficiency.

In contrast, reconfigurable architectures, such as CGRAs (coarse-grained reconfigurable architectures), have shown promise as highly efficient accelerator platforms [1]. They offer a smaller design space and simpler programming models than FPGAs (field programmable gate arrays), while still being able to create a customized datapath based on the target application or algorithm. As such, they potentially offer a middle ground between customization and programmability, but are constrained by their custom compilation flows and can not provide the simple programming model and flexibility of a GPU.

In this work, we take inspiration from CGRAs and propose the merging and sharing of resources amongst GPU cores (SMs) to better suit an application's need. Rather than allowing GPU SMs to run idle or with limited throughput, we propose shutting down SMs and re-distributing their resources to neighboring SMs. In doing so, we attempt to minimize the amount of power wasted, while limiting the impact to throughput.

In the past, coarse-grained reconfiguration of cores, [2] [3], has attempted to trade-off single-thread and multi-threaded performance. However, in this work we focus on targeting inefficient execution and improving energy efficiency by leveraging the GPU's latency hiding properties. We do so on a kernel-by-kernel basis and without the need for any ISA extensions or compiler modifications or any specialized CAD tools.

We summarize our contributions as:

- We explore the inefficiencies in GPGPU and show the unique requirements of individual kernels.
- Inspired by CGRAs, we propose inter-SM resource sharing and introduce four schemes to exploit it.
- We discuss the micro-architectural changes required within the SMs, along with their overheads.
- Our most aggressive scheme, BigSM, demonstrates up to 54% saving in energy, and an average of 26%.

The rest of the paper is organized as follows. In Sec. II we introduce our baseline GPU architecture and go on to motivate the problem in Sec. III. We then present our proposed solution in Sec. IV and define its micro-architecture in Sec. V. In Sec. VI we present our experimental evaluation, and follow it up with a discussion on CGRAs and related works in Sec. VII before concluding in Sec. VIII.

II. GPU Architecture

We consider a generic GPU model that consists of several *streaming multiprocessors* or SMs, along with a unified L2 cache. The SMs are interconnected to each other and the L2 cache via an on-chip interconnect. Fig. 1 illustrates the micro-architecture of the SM modeled and used in this work [4]. The SM manages threads in groups called warps. These warps are created, managed, and scheduled by the SM. By having a large number of *warps in flight*, the SM is able to mask the latency of memory access by swapping warps in and out of the pipeline, overlapping the execution of threads with memory accesses. This is done by maintaining the *context* of a warp on the SM. In order to do so, the SM features a large instruction buffer, multiple program counters (PC) and re-convergence stacks. Fetch, decode and issue are part of the scalar front end, such that instructions are fetched on a per-warp basis and stored in the instruction buffer after being decoded, while issuing of instructions is done in-order

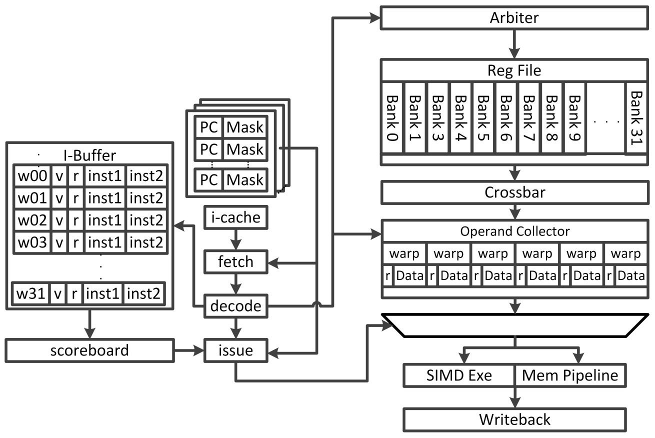


Figure 1. The streaming multiprocessor.

and branches are handled via predication and masks rather than via dedicated branch units and predictors.

As shown in Fig. 1, the back-end is comprised of a large and banked single-port register file, along with operand buffering via *operand collectors*. This register file is used to feed a wide SIMD execution unit where threads in a warp execute in lock-step. In addition, the SM provides a dedicated pipeline for memory operations, that interact with the local memory (data caches and shared memory) of the SM as well as global memory.

At first glance, CGRAs have a great deal in common with GPUs; they both have a large number of processing elements and simple control logic [1]. However, GPUs have well defined programming models, high bandwidth off-chip memory, complex memory hierarchies, and are not constrained to any particular domain.

III. Inefficiencies in GPGPU

In order to motivate the problem and to demonstrate the uniqueness of kernels, in this section we study a set of applications from the Rodinia [5] and Parboil [6] benchmark sets, listed in Table I. Our baseline SM and GPU configuration is described in Table II.

First, we analyze the number of registers needed per thread (REG), the amount of shared memory (SH MEM) needed per CTA (cooperative thread array or *thread block*), and the max number of warps per SM (THREAD) to determine the maximum number of warps (threads) that can concurrently reside on an SM i.e. *occupancy*. We present a summary of the benchmarks utilized, and list out the resource that constraints each kernel in Table I. From this *static analysis*, we observe how kernel requirements vary based on just three parameters: shared memory, registers, threads per SM. In addition, each kernel can be considered as compute or memory bound [7] [8], and may be constrained by algorithmic factors such as non-coalesced memory accesses and control divergence. We then further analyze the dynamic performance of the kernels, Fig. 2. First, we examine the throughput of each SM via IPC,

Table I
Benchmark applications and their resource constraints.

Application	Kernel Name	SM Resource Limit
Breadth First Search	BFSP	SH MEM,REG
Coulombic Potential	CUTCP	SH MEM
Histogram	HISTO1	THREADS
	HISTO2	REG
	HISTO3	THREADS
	HISTO4	SH MEM,REG
Lattice-Boltzmann	LBM	REG
MRI	MRI1	REG
	MRI2	THREADS
Dense Matrix Multiply	SGEMM	REG
Back Propagation	BP1	REG
	BP2	THREADS
Breadth First Search	BFSR1	THREADS
	BFSR2	THREADS
CFD Solver	CFD1	REG
	CFD2	REG
	CFD3	REG
	CFD4	THREADS
Gaussian Elimination	GAUS1	THREADS
	GAUS2	THREADS
Heartwall	HW	SH MEM,REG
Hotspot	HS	REG
Kmeans	KM1	THREADS
	KM2	REG
LU Decomposition	LUD1	SH MEM
	LUD2	SH MEM
	LUD3	THREADS
PathFinder	PATH	THREADS
SRAD	SRAD1	THREADS
	SRAD2	THREADS
	SRAD3	REG
	SRAD4	REG
	SRAD5	THREADS

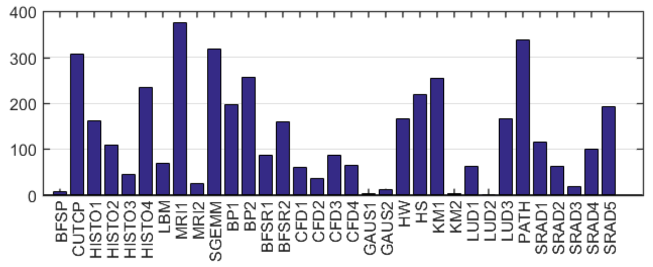
Fig. 2(a). It is evident that several kernels are not utilizing the GPU's full potential. In fact, out of the 33 kernels represented, only 10 kernels achieve an IPC greater than 50% of the theoretical peak, while only 17 kernels crossed the 25% mark.

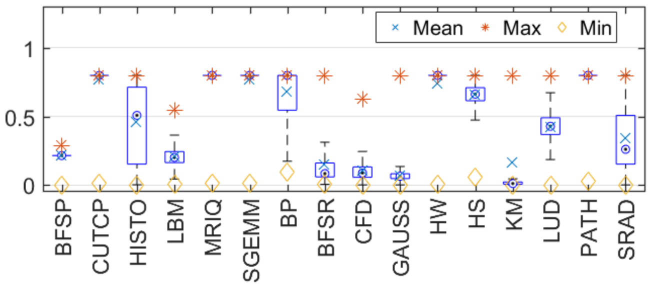
Fig. 2(b) and Fig. 2(c) show pipeline activity and power via box-plots that describe their variation across time in a statistical fashion. The target (circle with a dot in the middle) indicates the median value, while the edges of the box are the first and third quartiles of the data. Note that pipeline duty cycle is measured as the ratio of committed number of instructions to the maximum peak of committed

Table II
Architecture parameters used for simulation

Per Configuration SM Resource					
Resource	Baseline	Occ.	Bal.	Big SM	Big SM_Occ.
#SM	12	6	8	6	6
Warp Size	32	32	32	64	64
#Warps	24	48	36	24	48
#Reg	16K	32K	24K	32K	32K
#Reg Banks	16	32	24	32	32
Sh Mem	16K	32K	24K	32K	32K
Sh Mem Banks	16	32	24	32	32
L1 D$	16K	32K	24K	32K	32K
#Op Coll	16	32	24	16	16

GPU Resources			
L2 D$	#Mem Cntrl	Max #CTA Per SM	Core Clock
256K	4	8	700MHz


(a) Instructions Per Cycle (IPC)


(b) Pipeline Activity

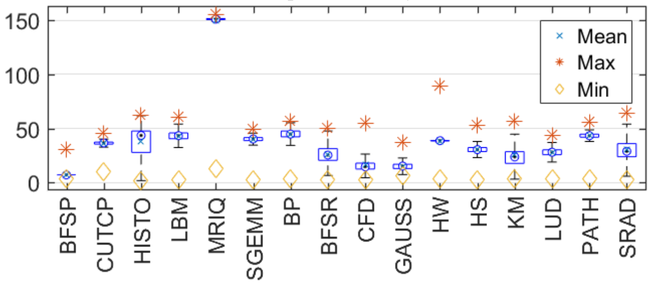

(c) Kernel Power

Figure 2. Kernel execution profiles.

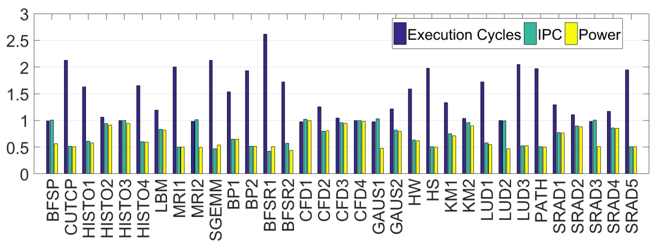


Figure 3. Kernel performance and power with half the SMs active, relative to the baseline configuration.

instructions [9]. It provides us with an insight into how much work is being done by the SM pipeline and is useful in quantifying the effect of stalls and idling on execution performance. A higher value indicates a more efficient usage of the core pipeline resources. From the plots we note that kernels within an application can demonstrate vastly different characteristics which can cause applications to appear to have variations across time in *phases*. However, these *phases* are aligned with different kernels within an application.

IV. Coarse Grained GPU Reconfiguration

Following our analysis in Sec. III, we consider shutting down half of the SMs as a first order solution to conserve power. In Fig. 3, we see that several kernels maintain the same IPC despite having half the number of execution units available. In fact, these kernels also showed poor IPC in our baseline analysis (Fig. 2). Thus, we make two observations: (1) Kernels in an application may have unique characteristics and resource constraints; (2) Increased computational resources may not provide improved throughput.

We propose a solution to address these inefficiencies in GPU computing - configure the GPU to suit the characteristics and requirements of the kernel prior to launching the kernel. Unlike CGRAs, we will not rely on any modifications to the existing code or any compiler optimization. Instead, we rely on merging and re-distributing three types of resources: (1) Thread context management, i.e. instruction-buffers, convergence stacks, PC stacks; (2) On-chip memory i.e. register banks, caches and shared memory; (3) SIMD execution units. Using a combination of these three parameters we present four schemes - Occ, Bal, BigSM and BigSM_Occ. Note that since each SM is very large, we limit the sharing of resources to neighboring SMs only.

A. Increasing Occupancy

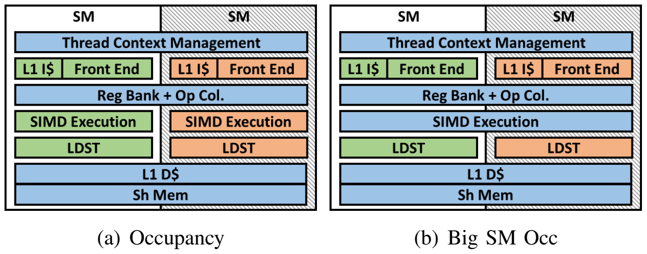
In Sec. III, we noted two possible causes of inefficient execution - memory latency and resource limitation. Since computation is not the bottleneck, we propose turning-off half the SMs on the GPU, and make their thread management resources available to the neighboring SM, thereby doubling it's occupancy and improving latency tolerance. In addition, to support the expanded context, we propose *merging* the register files as well as the shared memory and L1 data cache if needed. This configuration is of particular significance to memory bound applications. Table II includes the details of this configuration (Occ.) and is illustrated in Fig. 4(a).

B. Balanced Redistribution

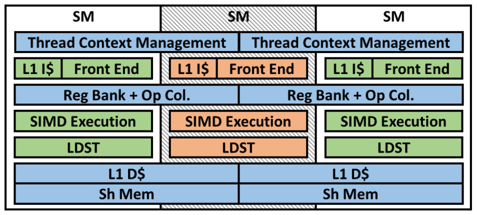
We also explore a more conservative approach. Rather than shutting down half the SMs, we shutdown only one-third, and distribute the context management and on-chip resources equally across the remaining SMs. Thus, we maintain two-thirds of the compute units active, while increasing the resident warps, registers and cache size by 50%. This *balanced* (Bal.) scheme is shown in Fig. 4(c) and detailed in Table II.

C. SIMD Expansion

Our most aggressive scheme proposes the merging of SIMD execution resources along with the increased context and memory. In doing so, we double the warp size which can potentially improve memory sub-system behavior. Since the SM manages work at the warp-level, by doubling the number of warps per SM and the size of each warp, the total number of threads per SM (occupancy) increases by a factor of 4. This, however, can lead to contention which may diminish the effect of improved occupancy. Thus, we explore two versions of this configuration: (1) BigSM, which only expands the warp size and (2) BigSM_Occ., which expands the warp size and the number of resident warps. This configuration is shown in Fig. 4(b) and detailed in Table II (BigSM).



(a) Occupancy (b) Big SM Occ



(c) Balanced

Figure 4. Configuration schemes.

Fig. 4 shows a high level overview of our schemes for a given pair or trio of neighboring SMs. The grey shaded SM is inactive in the given group, and has some of it's units *turned off* (orange shaded). The SM which is active, borrows resources from its inactive neighbor, and merges them with its own (blue shaded units). Other components in the active SM remained unchanged (green shaded units). An SM is considered *turned off* or inactive, when no warps are resident and executing on it's front end. The turned-off units are considered to be power-gated.

V. Micro-architecture of a Flexible SM

In order to enable flexible configuration of the GPU's SMs, global control structures as well as the SM must be redesigned. We introduce modifications such that the baseline performance is maintained, while the area and power overheads of the additional structures are minimal. In our discussion, we call an SM whose front-end is active and using resources from a neighboring SM, the *Active SM*, while the SM whose front-end is disabled and whose resources are being utilized by another SM, is called the *Inactive SM*.

A. Dummy Pipelining and Data Movement

Reading and writing from data structures across SM boundaries may not be feasible within a single cycle due to wire delay. For an SM reading operands from local and remote structures, this can create an imbalance in operand latencies. Rather than introducing complex control logic, we propose *dummy pipelining*, whereby we add dummy pipeline stages to local accesses. While this deepens the pipeline, and potentially increases instruction latency, GPUs are throughput oriented architectures and are designed to tolerate latencies. Fig. 5 illustrates the idea of dummy pipelining. Note that the *dummy* nomenclature refers to the two stages added between A and B, as they do not improve the timing characteristics and only balance the two paths. Additionally, a *bypass path*, that bypasses the dummy stages, can be used to revert to the baseline pipeline timing, should a kernel be better suited to the baseline architecture than our schemes.

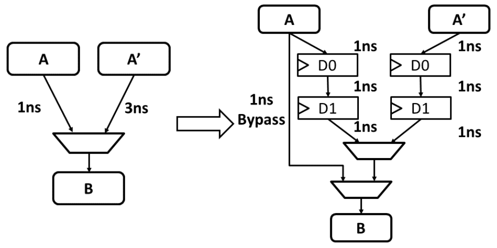


Figure 5. Dummy Pipelining.

For the rest of this discussion we will rely on *dummy pipelining* being implemented with the help of two elements: (1) Boundary buffers, placed at the edge of the SM to facilitate data movement across SM boundaries. (2) wire links, such as those demonstrated in [10] that enable low latency communication of small data packets and signals across cores. This is in contrast with CGRAs that rely on uniform routing networks between processing elements (PEs).

B. Enabling SM Resource Sharing

GPUs employ a global scheduling engine to assign CTAs from a kernel to each SM. In our architecture, we assume that a hardware register informs the scheduler how many SMs are available. At kernel launch, configuration bits are passed when the work queues are pushed to the GPU. These configuration bits set which SMs are the *Active SMs* and set the appropriate *shared resource mode* bits for each SM. When a CTA is scheduled to an SM, the SM is provided with configuration bits to enable/disable resource sharing. Thus, scheduling of CTAs is still handled by the hardware and does not require additional driver or compiler assistance, apart from providing configuration bits.

1) Front End: In our proposed architecture, we begin by partitioning the instruction buffer and convergence stacks into two banks, such that two concurrent reads may be done from two different banks. When an SM is in *shared resource mode* and is an *inactive sm*, then a copy of the top entry from the convergence stack is made available in a boundary buffer (Sec. V-A). By doing so, the scheduler of the *active SM* has fast access to it. If the *inactive SM's* resources are shared by two *active SMs* (Bal. config), then the two banks of the instruction buffer and the convergence stacks are used independently, with one bank being allocated to one *active SM*. Else, both banks can be used as a single structure by either the SM they are local to, or by one *active SM* only (Occ or BigSM configs).

Next, we propose adding two bits to each instruction buffer, so that they can hold the valid and ready bits for an additional warp, instead of just one warp. The active SM can then perform scheduling and scoreboarding without having to cross the SM boundary. However the scheduler and scoreboard logic needs to be able to handle twice as many warps. Finally, upon completing decode, the instructions need to be placed in the instruction buffer which may be in the *inactive sm*. By using the boundary buffers and dummy pipelining, we can tolerate the additional latency by pipelining the decode unit.

2) Datapath: The datapath is comprised of a register file, operand collectors and SIMD execution units. When modifying the datapath, our goal is to limit data movement between SMs and avoiding complex wiring and crossbar structures.

The register file is banked and capable of servicing requests to all banks simultaneously. Thus, we propose sharing of the register file on a bank-level, by assigning either half or all the banks to *active SMs*. However, operands are buffered in operand collectors before dispatch, which makes sourcing operands across SM boundaries prohibitively expensive. Instead, we propose a change in allocation policy, such that all warp operands, which may be in different register banks, and their operand collector are always in the same physical SM, thereby eliminating the need for a complex cross-SM crossbar. Since register indexing is done based on its index, we cannot ensure that all operand of a warp will be on the same SM without renaming logic or compiler support. So, we also modify the register indexing policy and constrain warps with indices higher than that of the baseline to use registers from the *inactive SMs*.

In the case of SIMD expansion (BigSM), wider warps require twice as many operands to feed the execution units. However, each register bank entry and each operand collector entry, is designed to hold operands for the standard warp size only. To tackle this, we mirror the register and operand requests across the two SMs, such that registers and operand collectors on each SM provide operands for half of the *wide warp*, giving the appearance of twice as many registers per cycle but the same number of register file banks and the same number of operand collectors. Upon execution, SIMD units are fed by operand collectors local to the SM.

In the case of Occ. and Bal. schemes, operand dispatch may require moving data across SMs. Again, we leverage *dummy pipelining* to transport data across SMs via boundary buffers. Thus, the operand dispatch is now pipelined. The first stage reads the operand to be issued, and places copies in the boundary buffers, while the second stage selects operands either from local collectors or from neighboring SMs via the boundary flops.

3) Memory Pipeline: The LDST unit operates as an independent memory pipeline and includes coalesce logic, memory request queues, address generation unit (AGU) and the network interface to the global interconnect. Sharing or merging these resources is not feasible due to their size and complexity. Thus, the external memory bandwidth per SM remains the same across all four schemes. In our four schemes, we provide for increased cache and shared memory capacity while relying on the *active SM's* LDST unit. Since shared memory is implemented as a multi-banked structure, where each bank can be independently accessed, we provide increased shared memory to the *active SM* by allocating shared memory banks from an *inactive SM* by mapping the additional banks to higher indexed memory addresses. The *active SM's* LDST unit is still responsible for generating the requests.

However, sharing cache resources is not straightforward. In order to share the cache, the tag arrays of a single way can be partitioned into two banks such that it can now service twice as many requests. When operating in the baseline configuration, the MSB of the address can be used to determine which bank the tag belongs to and local requests can be serviced. When operating in the *shared mode*, each bank of the tag array is dedicated to a remote SM. A selector bit, informs the tag lookup-logic that two external and independent requests must be serviced, which in turn ensures that the appropriate address is routed to the correct bank. If the entire cache is allocated to the neighboring SM (Occ. and BigSM), then the cache can operate as normal but services remote requests only. Assuming that cache accesses are multi-cycle operations, the next stage after tag comparison involves reading/writing into the selected set. Once again, in order to support the sharing of the cache, the data array is also banked to support two different requests. Once data is read, it is buffered in the boundary buffers, to assist in moving the cache line to the neighboring SM. Since requests to the remote memory banks will have a higher latency, we again leverage dummy pipelining and increases the overall latency of the LDST unit for shared memory and L1 cache accesses.

As in the case of the front-end and datapath, when operating in the default configuration, the memory pipeline maintains its original timing while the operating clock frequency is unchanged across all configurations.

4) Synchronization and Coherence: CTA-wide synchronizations are handled by the SM on which the CTA is resident and are handled by the front-end units only. In our proposed architecture, warp management is handled by the *active SM's*, hence we do not need additional mechanisms for CTA-wide synchronization. In addition, the register file, shared memory and L1 data cache resources are partitioned and allocated exclusively to an *active SM*, such that any given bank is never actively shared by two SMs simultaneously, which mitigates coherence issues. Finally, we perform configuration changes on a kernel-by-kernel basis, and since register and shared memory allocations are not valid across kernel launches, we do not need to introduce any additional flushing sequences.

VI. Evaluation

Table III
Static power breakdown per SM

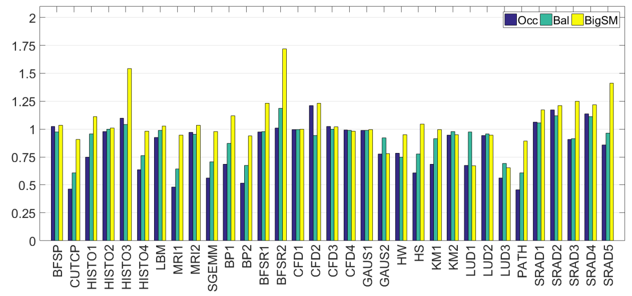
Unit	Leakage Power (W)
Front-end	0.06783
Reg File	0.228586
SIMD	0.038
LDST	0.911
Undiff	1.571
L1D$, Sh Mem	0.86
Tex, Const $	0.86
Additional Wiring	0.0385416

A. Overhead Analysis

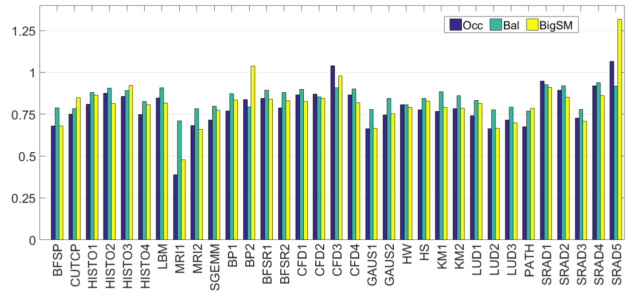
In order to estimate the area overhead of the modifications to the GPU, we use a combination of RTL synthesized models and estimates from Cacti [11]. First, we characterize the wiring overhead via Cacti [11] wire models, similar to [12]. We estimate the area of an SM, via GPUSimPow [13], to be $12.93mm^2$ in a 40nm technology, and thus its length is approximately $3.6mm$, which we use as the maximum distance data must be moved between two SMs. Since global wires are power hungry and have limited availability we use low swing wires. Low swing wires do not swing to full VDD, hence they consume less power, but are slower and need area consuming repeaters [11]. Using Cacti, we estimate the wire delay to be $0.73ns$ for $3.6mm$, which allows data transfer to be completed in a single cycle, with a 700MHz clock frequency. However, since our design uses *boundary buffers* as an intermediary, data must move only $1.8mm$, which would require $0.315ns$. Thus, the *dummy pipelining* overhead is a single cycle. Then, using a 45nm library and Synopsys Design Compiler, we estimate an 85% overhead to the issue unit, due to the pipelining, which accounts for $0.064um^2$ per SM. Additionally, we characterize the overhead of the boundary buffers and muxes as $0.0887um^2$, per SM. Finally, we estimate the total cost of the wiring assuming no overlap over logic, as done in [3], to be $4.71mm^2$ per pair of SMs, and the total leakage power of the wiring is $0.077W$, per SM pair. The total area of a 16SM GTX580 GPU is $520mm^2$ [13], in which case the wiring accounts for a 7.24% area overhead.

B. Evaluation Setup

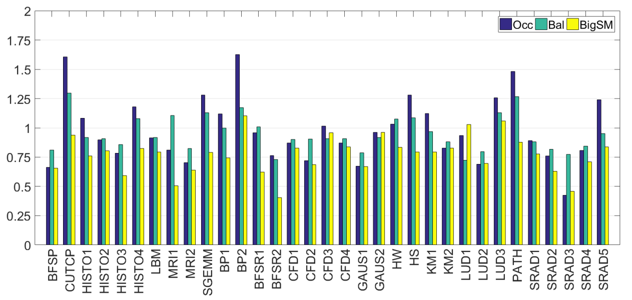
In this work, we use a cycle accurate simulator, GPGPU-Sim [4]. We model our GPU, based on the specs shown in Table II and modified it to reflect the pipeline timing and resource allocation policies discussed in Sec.IV. In order to estimate power, we use GPUWattch [9] to determine the dynamic power, and we use values provided by GPUsimPow [13] for detailed breakdown of static power. We also include the overhead of the wiring (Sec. VI-A), and use Cacti to calculate the leakage power for a unified L1D cache and share memory, as well as the texture and constant caches.



(a) IPC



(b) Power



(c) Energy

Figure 6. Experimental results, normalized to baseline configuration

C. Evaluation Results

Fig. 6 presents the results of our evaluation, normalized to the baseline configuration (Table II). The power data shown includes the static power for each configuration, based on the data in Table III, and energy is computed as the power-delay-product.

1) Experimenting with Occupancy: As outlined in Sec. IV, the configurations - Occ. and Bal., attempt to improve resource utilization by increasing occupancy, per-SM resources and save power by shutting down other leaky components that are not needed. Consider the kernels that exhibited very poor performance on the baseline GPU, such as BFSP, MRI2, GAUS1 and KM2. With the occupancy optimized scheme, they demonstrate less than 5% performance degradation. However, by shutting down wasteful units, we reduce the power consumption of these units by 33% to

14%, and thus provide 33% to 12% energy reduction across the Bal. and Occ. schemes. An interesting case is that of the register constrained CFD2 kernel which actually shows 20% increase in performance on Occ, but not on Bal. This is an artifact of our register allocation policy, where the remote SM's registers are available only to the additional warps (Sec. V-B2). Overall, the Occ. scheme shows an average of 15% reduction to IPC, while Bal. shows an average IPC reduction of 9%. However, this average includes compute intensive kernels, whose performance is expected to fall in these schemes. If we focus on kernels that do not degrade severely in performance, the Occ. scheme is able to provide up to 57.5% reduction in energy, with an average of 20%. On the other hand, the Bal scheme provides up to 27.5% reduction in energy, with an average of 14% only.

2) Warp Size Expansion: The results in Fig. 6 show promise for the BigSM configuration. Note that in BigSM, since the number of LDST units is halved, each SM now has to service twice as many threads in a warp, which can cause contention on the network and the memory. This is where increasing the occupancy along with widening the warp size (BigSM_Occ.) can be leveraged. Fig. 7, shows the relative IPC and energy of BigSM_Occ, over BigSM. Consider the compute bound kernels which per-

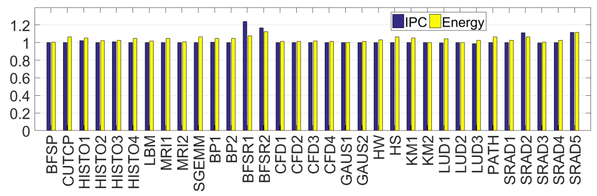


Figure 7. IPC and energy of BigSM_Occ. relative to BigSM

formed poorly on Occ. - CUTCP, MRI1, SGEMM, BP2, HS, KM1, PATH and SRAD5. On BigSM, they exhibit significantly better performance, relative to Occ. However, the impact of increasing occupancy along with SIMD width is minimal, as seen by the results of BigSM_Occ. In Sec. IV, we noted that increasing the warp size and the number of concurrent warps would quadruple occupancy. However, in our experiments, this increased occupancy was hard to saturate by the workloads, since the number of CTAs per SM did not change. Increasing the number of CTAs per SM is not trivial and would require modifying the global and local schedulers. An outlier to this is the BFSR2 kernel. The BFSR2 demonstrates a high IPC, however upon inspection, we see that it is primarily comprised of writes to global memory, with no compute. Hence it is able to leverage the wider warp execution, as well as the increased occupancy provided by BigSM_Occ.

As expected, BigSM performs well with compute intensive kernels, and is able to minimize the performance degradation when shutting down SMs. Additionally, occupancy limited kernels (Table I) such as those in BFSR and SRAD, show increased IPC due to the increase in available threads and registers. As we saw for the Occ. scheme, CFD2 is able to achieve over 20% speedup from the increase in registers in BigSM too. Overall, BigSM and BigSM_Occ demonstrate 5.8% and 8.6% increase in IPC across all kernels, and have a maximum IPC increase of 54% and 57% respectively (excluding BFSR2, which shows upto 100% increase in IPC for BigSM_Occ). When the power savings are taken into account, BigSM shows an average of 26% improvement in energy, and up to 54% energy reduction.

3) Choosing The Best Configuration: From our analysis and results, we can conclude that for memory bound kernels, the Occ. scheme is best suited. While, BigSM can provide similar performance, the additional write-back and data movement penalty is not justified. On the other hand, for compute bound problems, BigSM has been shown to provide the best energy efficiency and performance. In the case of where the kernels are constrained by resources, such as registers or shared memory, the decision must be made based on whether it requires additional occupancy and whether it is compute intensive or not.

VII. CGRAs and Related Works

When compared to GPUs, the custom CAD and compilation flow of CGRAs puts them at a disadvantage, despite their efficient customizable datapaths. In this work we have attempted to bridge this gap. Due to the difference in workloads, benchmarks, and evaluation methodologies, it is hard to perform a one-to-one comparison between CGRAs and our architecture. Instead, we use performance-per-watt (GFLOPS/W) for matrix multiplication as a relative comparison point of efficiency, and consider two works [14] [15]. [14] demonstrated 23.48 GLFOPS/W for SGEMM and we calculated the efficiency of [15] as 10.11 GFLOPS/W. In contrast, we measured 1.76 GFLOPS/W on our baseline GPU model, which was based on the Nvidia 45nm Fermi architecture. Note, [14] reported 0.77 GFLOPS/W for a GTX9800 GPU, which is one generation older than Fermi. However, our BigSM configuration was able to show an improvement of 26% and demonstrated 2.2 GFLOPS/W. Note, that our power estimates include the off-chip memory (DRAM) power as well. Clearly, further work is needed before GPUs reach the efficiency of CGRAs. However, as we demonstrated, by leveraging techniques from CGRAs, we significantly boost the execution efficiency of GPUs. Also, GPUs are more adept at handling memory-bound applications than CGRAs.

When compared to previous work, [2] and [3] are similar in spirit to our work but focus on CPUs and performance. The TRIPS architecture [2] leveraged coarse-grain reconfiguration to target three levels of parallelism: instruction, data, and thread level, while, core fusion [3] demonstrated a trade-off between instruction and thread level parallelism by proposing the *fusion* and *defusion* of several cores to form wider-issue cores. With respect to GPUs, [12] attempted to improve energy efficiency of GPUs by grouping SMs into clusters and sharing a front-end among them via local NoCs. Our work distinguishes itself by providing a more closely coupled architecture, without the overhead of an NoC and without the need to scale the scheduler, issue and fetch bandwidth. We also do not target runtime reconfiguration, and do not require any custom ISA support. In addition, our work derives energy savings by not only saving front-end power, but also back-end power, by shutting down the back-end (Occ. and Bal.), while still maintaining total throughput. Additionally, in [7] the authors describe a configurable on-chip memory that can be configured and redistributed as registers, shared memory and cache. In our work, we not only look at memory, but also try to address thread occupancy and SIMD execution. Finally, [16] performs a similar exploration in resource sharing, but from a performance perspective.

VIII. Conclusion

In this work, we studied the inefficiencies that can arise in GPU computing and demonstrated that kernels have diverse characteristics and resource requirements. We proposed a CGRA-inspired approach to share resources between the SMs of a GPU, thereby enabling it to adapt to the needs of the application and thus improve energy efficiency. By configuring the SMs, on a kernel-by-kernel basis, the overall energy efficiency of individual kernels, and therefore the application, can be significantly improved. Our evaluation showed that our most aggressive scheme, BigSM, is capable of achieving up to 54% savings in energy, and averages 26%.

While our Occ. scheme proved be more power conservative, we believe the Bal. scheme still shows promise. We hope further investigation will provide insight into how the Bal. scheme may be leveraged for dynamic load balancing and power management, by shutting down a few SMs for short periods of time to meet the TDP (thermal design power) requirements. With these results in mind, we believe that adopting cross-core resource sharing on GPUs is an exciting new opportunity and merits further study.

Acknowledgment

This work is supported by the IBM-ILLINOIS Center for Cognitive Computing Systems Research (C3SR) - a research collaboration as part of the IBM Cognitive Horizon Network. We would also like to thank Neal Crago from Nvidia for his valuable feedback and comments.

References

[1] Z. ul Abdin and B. Svensson, "Evolution in Architectures and Programming Methodologies of Coarse-grained Reconfigurable Computing," *Microprocess. Microsyst.*, vol. 33, May 2009.

[2] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, and et al., "Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture," in *30th Annual International Symposium on Computer Architecture, 2003. Proceedings.*

[3] E. Ipek, M. Kirman, N. Kirman, and J. F. Martinez, "Core Fusion: Accommodating Software Diversity in Chip Multiprocessors," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, 2007.

[4] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *2009 IEEE International Symposium on Performance Analysis of Systems and Software.*

[5] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and et al., "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC).*

[6] J. A. Stratton, C. I. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, and et al., "The Parboil Technical Report," University of Illinois, at Urbana-Champaign, IMPACT Technical Report 12-01, 2012.

[7] 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 *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture.*

[8] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron, "A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads," in *Workload Characterization (IISWC), 2010 IEEE International Symposium on.*

[9] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, and et al., "GPUWattch: Enabling Energy Optimizations in GPGPUs," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013.

[10] T. Krishna, C. H. O. Chen, W. C. Kwon, and L. S. Peh, "Breaking the on-chip latency barrier using SMART," in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA).*

[11] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0," in *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007).*

[12] T. Zhang and X. Liang, "Dynamic front-end sharing in graphics processing units," in *2014 IEEE 32nd International Conference on Computer Design (ICCD).*

[13] J. Lucas, S. Lal, M. Andersch, M. Alvarez-Mesa, and B. Juurlink, "How a single chip causes massive power bills GPUSimPow: A GPGPU power simulator," in *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS).*

[14] Z. E. Rkossy, D. Stengele, G. Ascheid, R. Leupers, and A. Chattopadhyay, "Exploiting scalable CGRA mapping of LU for energy efficiency using the Layers architecture," in *2015 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC).*

[15] Y. Kim and R. N. Mahapatra, "Hierarchical reconfigurable computing arrays for efficient CGRA-based embedded systems," in *2009 46th ACM/IEEE Design Automation Conference.*

[16] V. Jatala, J. Anantpur, and A. Karkare, "Improving GPU Performance Through Resource Sharing," in *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, 2016.