Dynamic Front-End Sharing In Graphics Processing Units

Tao Zhang and Xiaoyao Liang*
Department of Computer Science and Engineering
Shanghai Jiao Tong University, Shanghai, China, 200240
Email: {tao.zhang,liang-xy}@sjtu.edu.cn

***Abstract*—A modern GPU processor consumes several times power of a multi-core CPU and delivers a much higher processing throughput. Researchers propose various architectural innovations to improve its energy efficiency. We observe that different streaming processors (SMs) in a GPU tend to exhibit very similar behavior for many GPU workloads. If multiple SMs can be grouped together and work in synchronous manner, it is possible to save energy by sharing the front-end in the SM pipeline including the instruction fetch, decode and schedule units. For efficient flow control and program correctness, the proposed architecture can identify unfavorable conditions and ungroup the SMs when necessary. However, sharing pipeline front-end between multiple SMs brings architectural challenges. In this paper, we show our design, implementation and evaluation for such an architecture. Detailed experiment results manifest 33.7% front-end and 6.8% total GPU energy reduction can be achieved.**

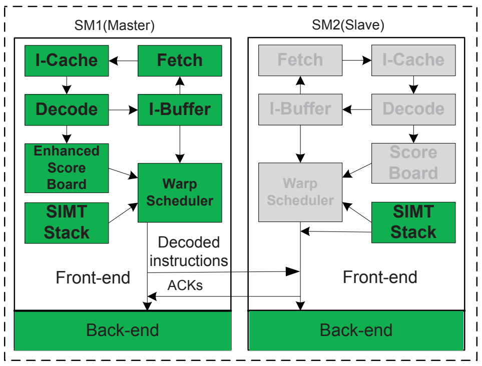


Fig. 1: Architecture of a front-end sharing cluster

I. Introduction

In recent years, GPUs have experienced tremendous growth as general-purpose and high-throughput computing devices. GPU vendors keep adding architectural innovations which push the parallel processing capability of a many-core GPU magnitudes higher than a multi-core CPU [5] [2]. On the other hand, a modern GPU chip consumes several times power of a CPU, therefore researchers propose various architectural solutions to improve the energy efficiency of GPUs [11] [8]. A typical GPU consists of multiple computing engines called streaming multiprocessors (SMs) as in Nvidia's terminology. The pipeline front-end in each SM for fetching, decoding and issuing instructions usually takes a significant portion of the transistor budget, and accounts for an average 18% of the GPU's total dynamic power [9]. Zhang et al. found that the fetch unit alone accounts for about 12% of the GPU power, and is the 4th most power-hungry component [21]. We are motivated to design architectural solutions for energy savings on the front-end.

In the GPU programming model (CUDA or OpenCL), each application has one or more kernels of multiple thread blocks. In the current GPU design, thread blocks are distributed across all SMs in a round-robin manner. SMs operate independently and the instruction execution flow is local to each individual SM. However, if thread blocks on different SMs come from the same kernel, the program instruction stream and the execution flow will be quite similar or sometime even identical for many GPU applications. This characteristic provides a unique opportunity for the resource sharing of the pipeline front-end among SMs, leaving room for power reduction in the fetch, decode, schedule and other units.

In this paper, we propose an architectural technique to share the GPU pipeline front-end among SMs for improved energy efficiency. Several adjacent SMs can be grouped into a sharing cluster for synchronous execution and one SM becomes the master. The master's front-end is always powered on while most components in the slaves' front-end are power-gated to save power. All SMs in a sharing cluster proceed under the instrument of the master. Different sharing clusters work independently without synchronization. There are associated challenges such as instruction issuing and the handling of branch divergences. We'll address these issues. In summary, this paper makes the following major contributions:

- We design and implement a novel pipeline front-end sharing architecture, which leverages the unique program execution behavior of thread blocks in the GPU. During the sharing period, only the front-end units in the master SMs are working while those of the slaves are powered off for energy savings.
- We carefully evaluate the front-end sharing architecture on a diversity of applications and two types of cluster formations. We analyze the performance and energy efficiency results.

II. Front-end Sharing Architecture

The front-end sharing architecture allows every S neighboring SMs in a GPU to form a cluster. In this paper, we show our results for two-SM cluster and four-SM cluster. The architecture of a two-SM cluster is illustrated in Figure 1. SM1 is the master while SM2 is the slave. In SM1, all the front-end components are activated. There is an enhanced scoreboard in

SM1. For memory instructions (eg. LD), the scoreboard tracks the data dependency for all cluster members since memory instructions vary in latency on different SMs. However, for non-memory instructions (eg. ADD, Multiply), it only checks the master's own data dependency since non-memory instructions have the same execution latency on all SMs. The warp schedulers of the master determine the instructions to be issued to the entire cluster. All SMs in a cluster work in a synchronous manner. There is a small Network on Chip (NoC) in the cluster for communications between cluster members.

In the slave SM2, all the front-end components are power-gated except for the SIMT stack. Slave SMs still manage their own SIMT stacks for recording branch divergences and reconvergence conditions. This is useful when the cluster is ungrouped after which SMs in the cluster will return to their independent operations.

A. *Instruction Issue*

One key question is how a master SM schedules and issues instructions. In each issue cycle, the master checks the criteria to decide if a warp instruction can be issued. Most cases it only needs to check its local information (SIMT stack, scoreboard, execution units status, etc.) because of the fully synchronized execution in the cluster, the datapath in all SMs will exhibit exact the same behavior. To issue an instruction to the slaves, the master needs to send decoded instructions through the NoC in the cluster. The details of the communications on the NoC is presented in section II-E.

However, there are exceptions for memory-related operations. Since the latency for accessing memory can be different across SMs, slaves are requested to send ”acks” to their master through the NoC to acknowledge the completion of memory accesses including the shared and global memories. After a master receives ”acks” from all the slaves, this instruction is cleared from the scoreboard. Non-memory instructions do not need acknowledgements because of the fixed latency. Each master uses an enhanced scoreboard to track the completion status of the memory instructions for the entire cluster. An enhanced scoreboard is implemented by adding four more bits to each scoreboard entry since it needs to track at most itself and three slave members in the cluster (for the four-SM cluster case).

B. *Uniform CTA Blocks Dispatching*

Each GPU has a global Concurrent Thread blocks (CTAs) scheduler for allocating thread blocks onto SMs. SMs work independently and can be assigned a different number of thread blocks in conventional design. However, to maintain the correct operation of front-end sharing clusters, the CTA scheduler needs to assign an equal number of blocks for all SMs in a cluster. Otherwise, the master may issue instructions not existing on slaves or some instructions on slaves never get issued. Either case will cause error. In our architecture, every SM in a cluster will receive an equal number of thread blocks at the beginning of execution. When all SMs in a cluster finish the thread blocks, the CTA scheduler will dispatch a new set of blocks onto them.

C. *Group*

At the time to launch a new kernel onto a GPU of N SMs, this GPU will be split into N/S clusters by grouping every S adjacent SMs into a cluster. SMs within a cluster are called cluster members. In each cluster, the SM with the minimal index becomes the master and all the rest SMs become slaves.

D. *Ungroup and Regroup*

The state-of-art GPUs utilize thread masks to distinguish the branch directions. At every branch, the threads in a warp going for one direction will set their thread masks to ”1”, while the others will set their masks to ”0”. In our front-end sharing architecture, the prerequisite for coupled execution is that the master and slaves in a cluster execute the same instruction flow. Nevertheless it is not always true even if SMs are processing thread blocks from a same kernel. When SMs in a cluster execute different paths, we call this **”SM divergence”**. The ”SM divergence” occurs more frequently in irregular applications.

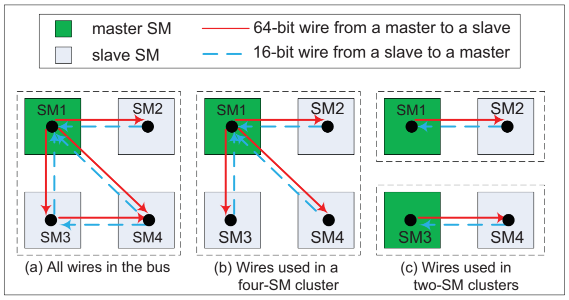
We evaluate a wide range of applications and most of them have no ”SM divergence”. For the cases like warps on SMs traverse all branch directions or take the same direction, this is not divergence and the front-end sharing cluster will not be affected. Besides, in almost all cases when ”SM divergence” occurs, the SM master and slaves will have different thread masks which can be easily identified. In some very rare cases, the ”SM divergence” happens even if SMs have the same thread masks. This is caused by the involvement of the thread ID into a condition evaluation so that the branch target addresses differ. To make our scheme simple and efficient, we leave these uncommon cases to the compiler. A compiler can easily identify such cases and signal the hardware using the compiler hint.

During execution, after a branch instruction has been executed (thus the thread masks are determined or the compiler hint is resolved), the master will broadcast its thread masks to all slaves in its cluster. All the slaves will compare their own thread masks with the master's. If they diverge, the slaves will send ”ungroup” requests to the master. The master will ungroup the cluster when it receives an ”ungroup” request from any of the slaves. After a ramp-down period for powering up the front-end of slaves and waiting for the master's scoreboard to be completely cleared, all SMs will run independently without the front-end sharing. Since the slaves keep their own SIMT stacks as shown in Figure 1, slaves once ungrouped will follow their own branch directions indicated in their local SIMT stacks. The ungroup/ramp-down takes place maximally once in every kernel.

Normally, GPU applications consist of multiple kernels with each implementing certain functions. Clusters once ungrouped will never regroup until the end of the kernel. At the beginning of the new kernel, SMs will have the opportunity to be grouped again even if they are just ungrouped in the last kernal.

E. *Communications in Cluster*

In the front-end sharing architecture, each cluster has a NoC for the communications between the master and slaves.



master SM
slave SM
64-bit wire from a master to a slave
16-bit wire from a slave to a master
SM1
SM2
SM3
SM4
(a) All wires in the bus
(b) Wires used in a four-SM cluster
(c) Wires used in two-SM clusters

Fig. 2: NoC in clusters

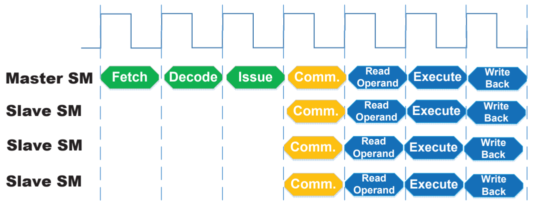


Fig. 3: Timing of pipeline stages

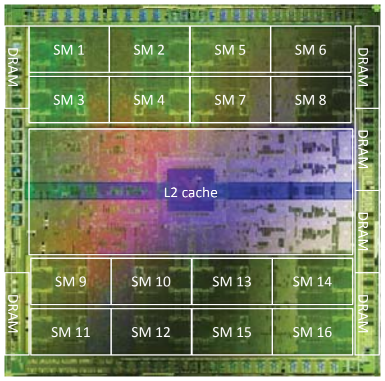


Fig. 4: Die photo of a GTX480 GPU

Figure 2(a) shows the connections. There is a pair of wires connecting the master and every slave. The connection from a master to a slave is 64-bit wide carrying the packets of decoded instructions. The connection from a slave to a master is 16-bit wide carrying the acknowledgements and other information. Figure 2(b) shows the wire connection for a four-SM cluster and Figure 2(c) shows the connection for the same cluster divided into two two-SM clusters. Same as the GPU interconnection network between the SMs and L2 cache, the NoC operates at twice the frequency of SM cores. However, the cluster NoC is totally 10 bytes (64-bit + 16-bit) wide, which is only $1/3$ of the width of the GPU interconnection network of 32 bytes.

There are three major types of packets on the NoC: $InstPacket$ contains instruction information; $MemPacket$ contains memory access "ack" message; $CtrlPacket$ controls the cluster behavior such as ungrouping or regrouping. CtrlPackets contribute a negligible portion of the total packets. Depending on the memory-intensiveness, MemPackets can take a significant portion of the network traffic. Below are the detailed information for each type of packets:

- $InstPacket$: This packet has 64 bits. The first 32 bits includes 6-bit warp ID, 4-bit function unit id, 6-bit operation ID and 16-bit immediate number. The second 32 bits contains five 5-bit registers IDs. If the instruction is a branch, there is an additional packet containing the master's thread mask.
- $MemPacket$: This packet has 16 bits. The slaves will acknowledge for the completion of each memory access with a packet. An "ack" packet has 2-bit access type, 3-bit slave ID, 6-bit warp ID and 5-bit register ID.
- $CtrlPacket$: Currently, there is only "ungroup" message sent in this type of packet. The packet has 1-bit type ID and 3-bit slave ID. The type ID is set to 1 for ungrouping.

Each Fermi SM has two warp schedulers. Therefore at most two instructions can be issued in each SM core cycle. Since the NoC operates at twice the frequency of the core, they have enough bandwidth to transfer two instructions at the core cycle. Figure 3 shows the timing of the pipeline stages for a four-SM cluster. Besides the regular pipeline stages, a new "communicate" stage is inserted between the issue and the read operand stage. The instruction transfer from a master to its slaves should be made in this stage. As shown in Figure 4, the Fermi GPU chip has roughly 23mmX23mm die size and is manufactured in 40nm technology [20] so we estimate the communication stage takes one-cycle latency to traverse the distance between two adjacent SMs which is roughly 5mm long. The delay of 5mm wire is 0.3ns (3.3GHz) reported by CACTI6.0 [14]. We clock the NoC at 1.4GHz, which not only meets one cycle latency, but also provides headroom for low-swing voltage operation that causes 0.6ns wire delay and 0.1ns signal regeneration that still meets timing. Low swing for wired bus is a common technology when latency is not critical. CACTI6.0 reports 50fJ/bit/mm for low-swing, about 1/6 energy of the full-swing. Transferring 64-bit instruction requires 50*64*5=16pJ, much less than reading 64-bit instruction from a 4KB I-cache incurring 32pJ by CACTI6.0. Furthermore, the slaves power off I-caches saving leakage power. The I-cache miss power also reduces because slaves never fetch instructions from main memory. The total area of the cluster NoC is estimated to be 2.3% of the area of the GPU interconnection network between SMs and L2 cache. The estimation is calculated based on the number of links (wires), the width of the links and the length of these links.

III. Experiment Methodology

A. Cluster NoC Power

We model the power of the cluster NoC using GPUWattch [13]. GPUWattch is an integrated power modeling tool in the GPGPU-Sim simulator [1] that can report the runtime power of each component. The Fermi architecture we simulated has a crossbar interconnection network for the communications between SMs and the L2 cache. Figure 4 shows the die photo [17] of a Fermi GPU. We marked the regions of each SM, L2

TABLE I: Simulator architectural configuration

Configuration items	**Value**
Shaders (SMs)	16
Warp Size	32
Capacity / Core	MAX. 1536 Threads, 8 CTAs
Core / Memory Clock	700 MHz / 924 MHz
Interconnection Network	1.4 GHz, 32 bytes wide, crossbar
Registers / Core	32768
Shared Memory / Core	48KB
Constant Cache / Core	8KB, 2-way, 64B line
Texture Cache / Core	4KB, 24-way, 128B line
L1 Data Cache / Core	32KB, 4-way, 128B line
L1 I-Cache / Core	4KB, 4-way, 128B line
L2 Cache	64KB, 16-way, 128B line
warp scheduler	Greedy then Oldest(GTO)
DRAM Model	FR-FCFS memory scheduler, 6 memory modules

cache and DRAM controller as indicated in [3]. We can see that the middle part of the chip is L2 cache with the cache controller at the center. At either side of the L2 cache, there are eight SMs closely fitted. Two and four memory controllers are located on the left and right sides of the chip, respectively.

We operate the cluster NoC at 1.4GHz, twice of the 700MHz SM frequency. This is the same speed as the interconnection network. However, they differ in the number of links, the average packet traverse distance, the NoC width and the amount of data transferred. We record the amount of data transferred for both the the cluster NoC and the GPU interconnection network at runtime. We also measure the average travel distance on Figure 4. We assume the power on the interconnection network and the cluster NoC is linearly proportional to the average distance and the amount of data transferred. Since GPUWattch can report the power consumption of the interconnection network, we can calculate the power of the cluster NoC.

B. Enhanced Scoreboard Power

We add four bits for each entry in the master scoreboard. 1-hot coding is used in the four bits to track the memory operations of the master and at most three slaves. We linearly scale the scoreboard power according to the ratio of the added bits over the original bits. This is due to the fact that the scoreboard mostly performs matching operations. One bit stands for one set of matching logics and the total power is proportional to the number of bits in the scoreboard. The area of these extended bits will be quite small and can be ignored.

C. Benchmarks

To show the generality of the proposed schemes, we mix benchmarks from various sources:

- NVIDIA CUDA SDK 4.1 [4]: BinomialOptions (BO), MergeSort (MS), Histogram (HG), Reduction (RD), ScalarProd (SP), dwtHarr1D (DH), BlackScholes (BS), SobolQRNG (SQ), Transpose (TP), Scan (SC).
- Parboil [18]: sgemm (SGE), Sum of Absolute Difference (SAD).

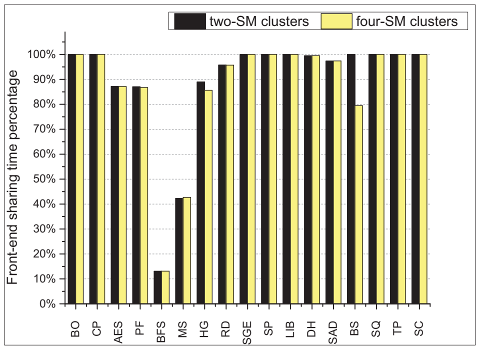


Fig. 5: Front-end sharing time percentage

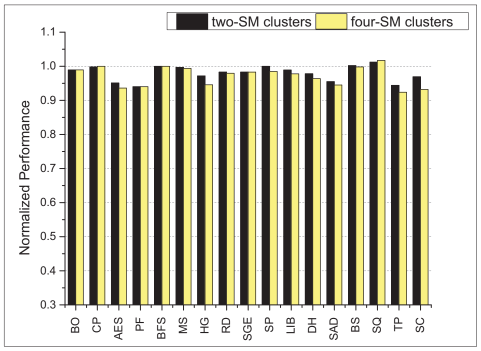


Fig. 6: Performance comparison

- Rodinia: PATH Finder (PF).
- GPGPU-Sim benchmark suite [1]: Coul Potential (CP), AES Encryption (AES), BFS Search (BFS), Swap Portfolio (LIB).

We select the applications for diversity: there are memory-intensive applications (BS, SQ, TP and SC) and compute-intensive applications (BO, CP, AES and PF), irregular (BFS, MS and HG) and regular applications (all the rest). In addition, most applications have multiple kernels, and each kernel can be the regrouping point.

D. Simulator Configurations

We use GPGPU-Sim 3.2.1 [1] as our simulation platform. We adopt the settings for the NVIDIA Fermi architecture. The machine parameters are listed in Table I. The default Greedy then Oldest (GTO) scheduler [16] is used as the warp scheduler. We evaluate the GPU and workload behavior under two-SM and four-SM clusters configurations. The runtime statistics including performance and power numbers are captured for each benchmark for analyzing. The power consumption of each component is acquired from the GPUWattch [13] integrated into the GPGPU-Sim.

IV. Results and Analysis

This section presents different applications' behavior under the front-end sharing execution.

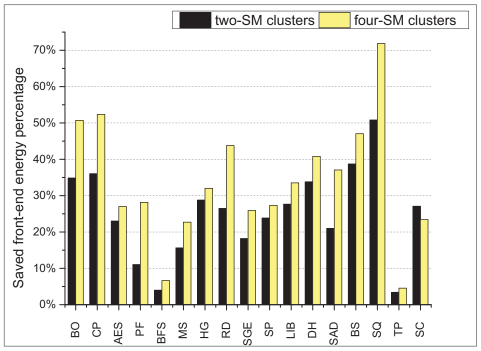


Fig. 7: Percentage of saved front-end energy

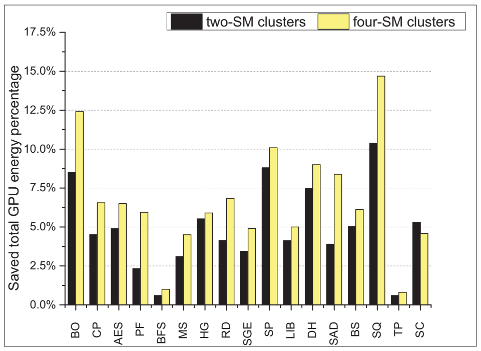


Fig. 8: Percentage of saved total GPU energy

A. *Percentage of Front-end Sharing Time*

Figure 5 shows the average percentage of cycles in the front-end sharing mode to the total execution cycles. A 100% means that all SMs are in front-end sharing execution during the application's entire lifetime. Many applications has no SM divergence, and therefore have 100% front-end sharing time percentage. Irregular applications such as BFS, MS, and HG all have sharing time percentage less than 100%. In general, the percentage is affected by how often and where the SMs diverge. Applications that have frequent SM divergences or that SMs mostly diverge at an early stage of kernels will have smaller sharing time percentage.

B. *Performance*

Figure 6 compares the performance of a two-SM cluster system and a four-SM cluster system. The result is normalized to the performance of a baseline GTX480 GPU without front-end sharing. Overall, we find the performance of the front-end sharing architecture is close to that of the baseline architecture.

Although our architecture groups several SMs for synchronized execution, it still keeps a warp size of 32 as the default. Our architecture can be flexibly configured and adapt to the varying thread level parallelism among applications by adjusting the cluster size and grouping/ungrouping SMs when necessary. In summary, the architecture achieves 98.0% and 97.1% performance on average of all applications for two-SM cluster and four-SM cluster, respectively. Our architecture is still much better for applications with enough thread level parallelism. In the default 32-wide-warp architecture, if all threads on an SM take one direction and all threads on another SM take a different direction, they can execute concurrently in our scheme. But grouping all the threads on a large-warp architecture (eg. fixed 64-wide warp or 128-wide warp), they will be serialized causing performance loss.

C. *Front-End Energy Savings*

Figure 7 presents the energy saved at the SM front-end for two-SM cluster and four-SM cluster configurations. The results correspond to the energy savings minus the overhead. The energy savings come from the power-gated front-end of the slaves in front-end sharing execution. The dynamic and static power of the front-end are acquired from the GPUWattch at runtime. The energy overhead includes the energy consumed on the cluster NoC, the enhanced scoreboard and the energy due to the prolonged execution time for some applications. Generally, larger front-end energy saving percentage can be achieved if an application has larger sharing time percentage and better performance. We can see that all applications save energy in both two-SM cluster and four-SM cluster cases. Generally, four-SM clusters save more energy because more slaves can be power-gated. It is worthwhile to mention that four-SM clusters have larger energy overhead due to the slightly reduced performance than two-SM clusters. Also, the energy spent on the cluster NoC is larger for four-SM clusters because the packets need to travel longer distances. On average, 24.9% and 33.7% energy can be saved under two-SM cluster and four-SM cluster configurations, respectively. We don't advocate cluster size beyond four because the performance/power is not good. SQ achieves the best front-end energy saving percentage because of its good performance and 100% sharing time. On the contrary, BFS and TP save only a poor portion of the front-end energy because of their small sharing time or bad performance.

D. *Total GPU Energy Savings*

Figure 8 presents the results for the total saved energy of the whole GPU. The energy saving percentage for the entire GPU depends on three factors: application performance (Fig. 6), sharing time percentage (Fig. 5), and the percentage of the front-end power in total GPU power. We can see that four-SM clusters saves more energy than two-SM clusters for all applications except SC. For SC, its worse performance in the four-SM clusters and the corresponding energy overhead outweigh the extra power saved from more slave SMs. Overall, SQ saves the highest percentage while BFS and TP save the least energy percentage. Three applications save more than 10% total energy. From the figure, we can see that compute-intensive applications (BO, CP, AES, PF), memory-intensive applications (BS, SQ, TP, SC), and some irregular applications (BFS, MS, HG) all benefit from the front-end sharing architecture for energy efficiency. On average, 4.9% and 6.8% total energy savings are obtained for all applications under two-SM cluster and four-SM cluster configuration, respectively.

V. Related Work

As far as we know, this work is the first to arrange several SM processors to work in lock-step manner in GPUs.

Combining several smaller cores into a single larger, more capable CPU core has been studied in previous work. Core fusion by Ipek et al. [10] and core federation by Tarjan et al. [19] were proposed where the cores of a homogeneous CMP were reconfigured at runtime into stronger cores by ”fusing” resources from the available cores. However, they both use non-centralized hardware and the scalability is limited by branch prediction, memory addresses, and instruction window size. Another approach to fuse homogeneous cores is presented as Composable Lightweight Processors [12], where 32 dual-issue cores could be fused into a single 64-issue processor. All these schemes [10] [19] [12] exhibit a high inter-core communication overhead. The Voltron architecture from Zhong et al. [22] allows multiple in-order VLIW cores of a chip multiprocessor (CMP) to combine into a larger VLIW core. The Cray X1 gangs together SSPs in an MSP for synchronous execution [6]. Our work does not try to combine SMs to improve overall performance, but instead attempt to power off most front-end units in GPUs and leave only one front-end in each sharing cluster to server all the back-ends to save energy.

Branch divergence decreases GPU performance, thus various approaches have been proposed. Fung et al. [7] proposed the dynamic formation of warps to deal with diverging branches. Narasiman et al. [15] proposed the large warp microarchitecture and the two-level warp scheduling. Their large warp architecture creates fewer but larger warps, and dynamically creates SIMD-sized sub-warps from the active threads in the large warps. Our work also deals with branch divergence, but for a different purpose and on the SM granularity.

VI. Conclusion

In this paper, we propose a front-end sharing architecture to improve the energy efficiency in GPU. Multiple adjacent SMs can be grouped together and execute in a lock-step fashion in a sharing cluster. The working principle is based on the unique characteristic in GPU that many thread blocks behave similarly during the program execution and thus there is no need for duplicated front-ends and independent operations. The architecture can save 6.8% on average and up to 14.6% of total GPU energy. The experiments show that this architecture is effective in both compute-intensives and memory-intensives applications. In addition, some irregular applications can also benefit from the proposed architecture for energy efficiency.

Acknowledgment

The authors would like to thank anonymous reviewers for their fruitful feedback and comments that have helped them improve the quality of this work. This work is partly supported by the National Natural Science Foundation of China (Grant No. 61202026 and No. 61332001) and Program of China National 1000 Young Talent Plan.

References

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

[2] N. Brookwood, “Amd fusion family of apus: Enabling a superior, immersive pc experience,” *AMD White Paper*, 2010.

[3] N. Corporation, “Nvidia’s next generation cuda compute architecture: Fermi,” *Nvidia White Paper*, 2009.

[4] N. Corporation, “Nvidia cuda sdk 4.1,” *https://developer.nvidia.com/cuda-toolkit-41-archive*, 2012.

[5] N. Corporation, “Nvidia’s next generation cuda compute architecture: Kepler gk110,” *Nvidia White Paper*, 2012.

[6] T. H. Dunigan Jr, J. S. Vetter, J. B. White III, and P. H. Worley, “Performance evaluation of the cray x1 distributed shared-memory architecture,” *Micro, IEEE*, vol. 25, no. 1, pp. 30–40, 2005.

[7] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt, “Dynamic warp formation and scheduling for efficient gpu control flow,” in *IEEE/ACM 45th Annual International Symposium on Microarchitecture (MICRO)*, 2007, pp. 407–420.

[8] N. Goswami, B. Cao, and T. Li, “Power-performance co-optimization of throughput core architecture using resistive memory,” in *International Symposium on Computer Architecture (ISCA)*, 2013.

[9] S. Hong and H. Kim, “An integrated gpu power and performance model,” in *Proc. of International Symposium on Computer Architecture (ISCA)*, 2010, pp. 280–289.

[10] E. Ipek, M. Kirman, N. Kirman, and J. F. Martinez, “Core fusion: Accommodating software diversity in chip multiprocessors,” in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2007, pp. 186–197.

[11] N. Jing, Y. Shen, Y. Lu, S. Ganapathy, Z. Mao, M. Guo, R. Canal, and X. Liang, “An energy-efficient and scalable edram-based register file architecture for gpgpu,” in *International Symposium on Computer Architecture (ISCA)*, 2013.

[12] C. Kim, S. Sethumadhavan, M. G. Nitya, Ranganathan, D. Gulati, D. Burger, and S. W. Keckler, “Composable lightweight processors,” in *IEEE/ACM Annual International Symposium on Microarchitecture (MICRO)*, 2007, pp. 381–394.

[13] J. Leng, S. Gilani, T. Hetherington, A. ElTantawy, N. S. Kim, T. M. Aamodt, and V. J. Reddi, “Gpuwattch: Enabling energy optimizations in gpgpus,” in *proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2013.

[14] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, “Cacti 6.0: A tool to model large caches,” *HP Laboratories*, 2009.

[15] 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 *IEEE/ACM 45th Annual International Symposium on Microarchitecture (MICRO)*, 2011.

[16] T. G. Rogers, M. O’Connor, and T. M. Aamodt, “Cache-conscious wavefront scheduling,” in *2012 IEEE/ACM 45th Annual International Symposium on Microarchitecture (MICRO)*, 2012, pp. 72–83.

[17] T. Sandhu, “Nvidia’s geforce gtx 480 finally unleashed. reviewed and rated.” *http://hexus.net/tech/reviews/graphics/24000-nvidias-geforce-gtx-480-finally-unleashed-reviewed-rated/?page=2*, 2010.

[18] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W. mei W. Hwu, “Parboil: A revised benchmark suite for scientific and commercial throughput computing,” *IMPACT Technical Report*, 2012.

[19] D. Tarjan, M. Boyer, and K. Skadron, “Federation: Repurposing scalar cores for out-of-order instruction issue,” in *Design Automation Conference*, 2008, pp. 772–775.

[20] T. Valich, “nvidia ’fermi’ geforce die sizes exposed,” *http://www.brightsideofnews.com/news/2010/8/9/nvidia-fermi-geforce-die-sizes-exposed.aspx*, 2010.

[21] Y. Zhang, Y. Hu, B. Li, and L. Peng, “Performance and power analysis of ati gpu: A statistical approach,” in *6th IEEE International Conference on Networking, Architecture and Storage (NAS)*, 2011, pp. 149 – 158.

[22] H. Zhong, S. A. Lieberman, and S. A. Mahlke, “Extending multicore architectures to exploit hybrid parallelism in single-thread applications,” in *International Symposium on High-Performance Computer Architecture (HPCA)*, 2007, pp. 25–36.