Power-Aware L_1 and L_2 Caches for GPGPUs

Ehsan Atoofian and Ali Manzak

Electrical Engineering Department,
Lakehead University,
Thunder Bay, Canada
{atoofian,amanzak}@lakeheadu.ca

Abstract. General Purpose Graphics Processing Units (GPGPUs) employ several levels of memory to execute hundreds of threads concurrently. L_1 and L_2 caches are critical to performance of GPGPUs but they are extremely power hungry due to the large number of cores they need to serve. This paper focuses on power consumption of L_1 data caches and L_2 cache in GPGPUs and proposes two optimization techniques: the first optimization technique places idle cache blocks into drowsy state to reduce leakage power. Our evaluations show that cache blocks are idle for long intervals and putting them into drowsy mode immediately after each access reduces leakage power dramatically with negligible impact on performance. The second optimization technique reduces dynamic power of caches. In GPGPU applications, many warps have inactive threads due to branch divergence. Existing GPGPU architectures access cache blocks for both active and inactive threads, wasting power of caches. We use active mask of GPGPUs and access only the portion of cache blocks that are required by active threads. By dynamically disabling unnecessary sections of cache blocks, we are able to reduce dynamic power of caches significantly.

Keywords: GPGPU, CUDA, Memory hierarchy, Cache, Power.

1 Introduction

Early Graphics Processing Units (GPUs) exploited software-managed local memories (or scratch-pad) instead of caches. GPU workloads include large amount of streaming data which are difficult to cache. However, recent general purpose GPU applications demonstrate high level of data locality which makes them suitable for caches. In response, GPU vendors have included caches in their designs. For instance, NVIDIA introduced up to 48KB L_1 cache per core in Fermi [9] and AMD's Fusion GPU [13] offers 16KB L_1 cache per core. Both vendors' recent GPUs have global coherent L_2 caches. NVIDIA increased size of L_2 cache from 768KB in Fermi architecture [9] to 1536KB in GK110 [11]. It is expected that the size of caches grows in future.

Large caches consume significant static and dynamic power. This problem exacerbate in future: voltage reduction has slowed down in recent years, limiting dynamic power reduction through voltage scaling. Lowering the threshold voltage results in significant increase in static power. Therefore, it is necessary to optimize caches to reduce power consumption.

F. Silva et al. (Eds.): Euro-Par 2014, LNCS 8632, pp. 354–365, 2014.

Several architectural and circuit level techniques have been proposed to deal with the power of caches in processors [6, 14]. However, GPGPUs provide unique opportunities to reduce power of caches due to their architecture. For example, once a cache block is accessed by a thread, it takes several hundreds of clock cycles until the same block is accessed again. This is mainly due to the round-robin scheduling policy [5] used in GPGPUs. So, once a thread is executed, it should wait until GPGPU schedules other threads before it is executed again. The long inter-access delay can be used to reduce leakage power by placing cache blocks into drowsy mode [8] immediately after each access. The other opportunity for optimization of caches in GPGPUs is related to underutilization of cache blocks. Due to branch divergence, some applications are not able to fully utilize warp slots each cycle. Hence, dynamically disabling access to inactive cache blocks can reduce dynamic power.

In summary, this paper makes the following contributions:

1) The inter-access delay of L_1 and L_2 cache blocks is in the range of several hundreds of clock cycles. We exploit this property and propose a method that dynamically changes the state of cache blocks between ON and drowsy.

2) The number of active threads within a warp varies across the cycles. We exploit GPU active-mask feature to detect inactive portions of cache blocks before an instruction is scheduled for execution. We disable bit-lines, word-lines, and sense amplifiers of inactive SRAM cells to reduce dynamic power in L_1 and L_2 caches.

The remainder of the paper is structured as follows. Section 2 describes our baseline GPGPU model. Section 3 explains the motivation behind this work. Section 4 details our optimization techniques. Section 5 discusses our measurement methodology and reports the results. Section 6 describes related work and Section 7 concludes the paper.

2 Background

In this section, we provide a brief description of GPGPU architecture. For consistency, we use NVIDIA and CUDA terminology in this paper. However, our techniques are general and can be applied to a broader range of GPGPUs from other vendors.

A GPGPU consists of many Streaming Multiprocessors (SMs) and each SM typically has 8 to 32 Processing Elements (PEs). For instance, NVIDIA's Fermi series has 16 SM and each SM has 32 PEs. Figure 1 shows architecture of a GPGPU. Each SM is associated with a private L_1 data cache and read-only constant and texture caches along with a low latency shared memory. The memory is organized as several DRAM banks and each bank is associated with a slice of shared L_2 cache. SMs and L_2 cache are connected through an interconnection network. In this work, we use a 2D mesh topology for interconnection network since it is simple to implement and is throughput-effective [4].

A CUDA program is composed of one or more kernel functions that are launched and executed on the GPGPU (Figure 2). Each kernel divides its work into identically sized groups, called Cooperative Thread Arrays (CTAs). Every CTA is assigned to an SM for execution. To improve utilization of resources in an SM, more than one CTA

can be assigned to the SM. The maximum number of CTAs per SM is limited by SM resources such as number of threads, size of shared memory and register file, etc. [10]. For example if a CTA requires 8KB of shared memory and the baseline SM has 32KB available, then only 4 CTAs can be launched simultaneously on the same SM. From a programmer's point of view, all threads within a CTA execute each instruction in the kernel concurrently. However, on the real hardware, because of resource constrains, software threads are actually executed in groups of threads called warps. A warp has 32 threads on current NVIDIA GPUs. The SM executes one warp at a time. If a warp is stalled due to a long latency instruction, then the SM selects another warp for execution.

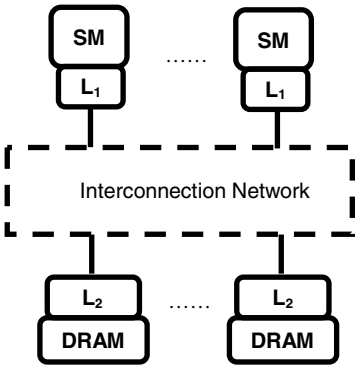


Fig. 1. GPGPU architecture

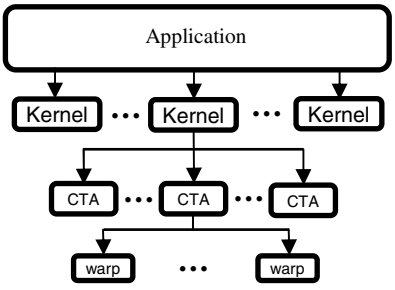


Fig. 2. GPGPU application hierarchy

A GPGPU kernel commonly accesses global memory space which is shared by all threads. When threads access data in the global memory, their accesses go through a two-level cache hierarchy. The L_1 caches are private to SMs but the L_2 cache is shared by all SMs. The L_1 caches are not coherent. They follow write-evict, write-no-allocate policy [10]. On the other side, the L_2 cache is coherent and uses write-back with write-allocate policy [10]. The cache blocks in GPGPUs are wide. For instance, in Fermi family, the cache blocks in L_1 and L_2 caches are 128 bytes. So, if all load or store instructions of a warp map to the same cache line, then all threads of the warp can be completed in a single transaction.

In this work, we employ a two-level scheduler [17]. The scheduler partitions warps into two groups: an active group holding warps eligible for execution and an inactive group of pending warps. Warps that are waiting for long latency events such as loads from DRAM are placed in the pending set. Once a warp is ready for execution, it is removed from the pending list and is inserted into the active list. This approach avoids stall cycles in a one level round-robin based scheduler [5] since warps progress with different speeds and the probability that all warps stall due to a long latency memory operation reduces significantly.

3 Motivation

In this section, we explain motivation behind our work and characterize several workloads used in this study to show power saving opportunities in GPGPUs. We use applications from NVIDIA SDK [18], Rodinia Benchmark suite [20], and Parboil Benchmark suite [21] (for detail of experimental framework, please refer to Section 5). The second column in Table 1 shows abbreviations for the benchmarks.

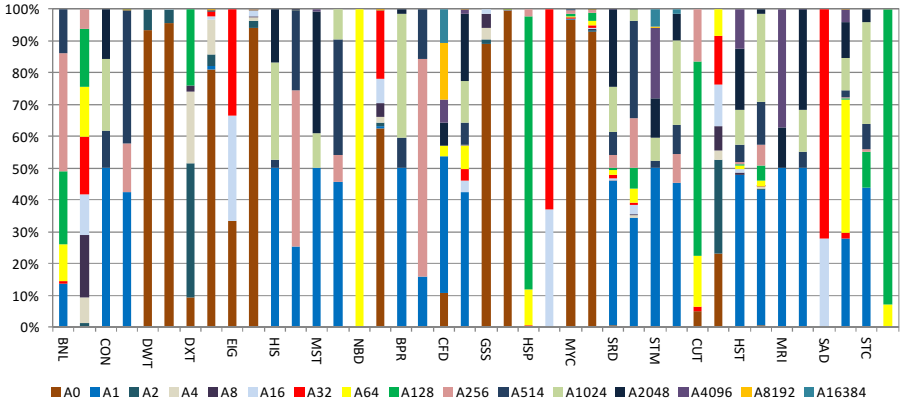


Fig. 3. Breakdown of accesses to cache blocks in L_1 and L_2 caches

Figure 3 shows breakdown of accesses to the cache blocks in L_1 and L_2 caches. For each benchmark, the first bar corresponds to the L_1 cache and the second bar corresponds to the L_2 cache. Each bar in the graph is divided into 16 sections. The top most component of a bar labeled A16384 shows number of blocks that are accessed 16384 times or more. Similarly, the bottom most component labeled A0 shows the number of blocks that are not accessed by any SMs. In L_1 caches, 50% of cache blocks are accessed 16 times or less. In L_2 cache, 50% of cache blocks are accessed 8 times or less. Since most of memory requests are serviced by L_1 caches, cache blocks in L_2 are idle more often. In DWT, GSS, and MYC, more than 88% of the cache blocks are never used for execution of the programs. These cache blocks can be put into drowsy mode to reduce power consumption.

Next, we focus on cache blocks that are accessed by PEs. Figure 4 shows breakdown of inter-access cycles for cache blocks in L_1 and L_2 caches. For each benchmark, the first bar corresponds to the L_1 caches and the second bar corresponds to the L_2 cache. We measure the number of cycles elapsed between two consecutive accesses to the same cache blocks. For L_1 caches, more than 50% of cache blocks have inter-access cycle of 128 or more. For L_2 cache, more than 50% of cache blocks have inter-access cycle of 64 or more. On average, the inter-access cycle for L_1 and L_2 caches are 2442- and 2840-cycle, respectively. In the two-level scheduler, after a warp is scheduled for execution, it should wait until all the other warps in the active list are scheduled. The only time that a warp is scheduled for execution in two consecutive cycles is when there is no other warp in the active list. Hence, quite often, there is a gap between two executions of a warp. This inter-access delay provides opportunity to put the cache blocks into drowsy mode immediately after they have been accessed.

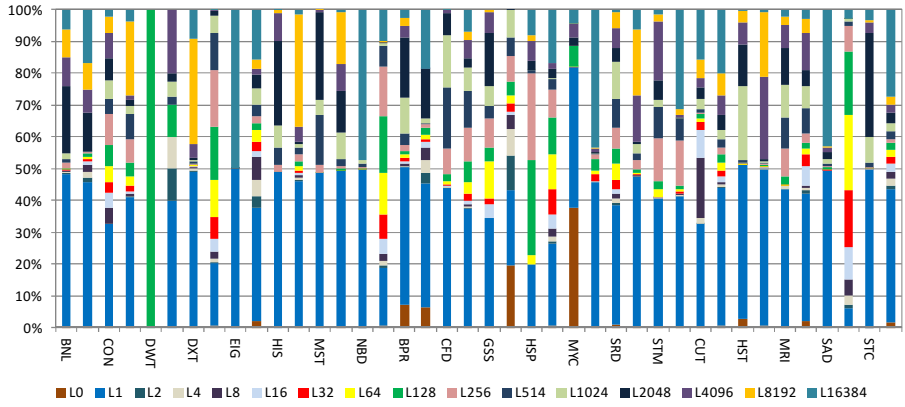


Fig. 4. Breakdown of inter-access cycles for cache blocks in L_1 and L_2 caches

GPGPUs execute threads in the granularity of warps. Each warp consists of 32 threads executing instructions in a lock-step manner. A fully utilized warp has 32 active threads executing one instruction at a time. In Graphics applications, quite often warps utilize all 32 slots. However, this may not be true for general purpose applications. General purpose applications exhibit more complex control flow behavior due to frequent branch instructions. Conditional branch instructions can cause threads within a warp take different paths, or diverge. Since GPGPUs allow a warp to have only one active PC at any given time, GPGPUs execute taken and not-taken paths in two phases. In the first phase, threads in the taken path execute and all threads in the not-taken path are idle. In the second phase, threads in the not-taken path execute and the rest are idle. Existing GPGPU implementations access cache blocks for all 32 threads within a warp although many warps may have fewer than 32 threads. The last two columns in Table 1 show the percentage of active threads within the warps that access L_1 and L_2 cache blocks, respectively. It is important to note

that L_2 cache is accessed when a miss occurs in any of the L_1 caches including data, texture, and shared L_1 caches. This is the main reason that block utilization in L_2 cache is lower than block utilization in L_1 data caches. While in some benchmarks, i.e. CUT, all warps have 32 active threads throughout the entire execution, some others, i.e. MYC, have very low cache block utilizations. Unnecessary accesses to the cache blocks in benchmarks with low warp utilization waste power. By avoiding these unnecessary accesses, we can reduce dynamic power in caches.

Table 1. GPGPU Benchmarks and Warp Utilization

Benchmark	Abbr.	L_1 block Utilization	L_2 block Utilization
binomialOptions	BNL	99%	98%
convolutionSeparable	CON	100%	81%
dwtHaar1D	DWT	100%	95%
dxtc	DXT	100%	1%
eigenvalues	EIG	100%	33%
histogram	HIS	100%	94%
MersenneTwister	MST	100%	73%
nbody	NBD	100%	14%
backprop	BPR	91%	81%
cfd	CFD	100%	93%
gaussian	GSS	65%	37%
hotspot	HSP	100%	96%
myocyte	MYC	4%	2%
srad_v1	SRD	99%	99%
streamcluster	STM	98%	99%
cutcp	CUT	100%	100%
histo	HST	100%	98%
mri-gridding	MRI	100%	99%
sad	SAD	93%	93%
sgemm	STC	100%	100%

4 Reducing Power of L_1 and L_2 Caches

In this section, we present static and dynamic power reduction techniques based on opportunities discussed in Section 3.

4.1 Static Power Reduction Using Drowsy Cells

Inter-access cycles in Figure 4 show that cache blocks are not accessed for long intervals and it is possible to save power of cache blocks when they are idle. Several techniques have been proposed to reduce leakage power of cache cells by turning off cache blocks when they are not needed [1, 22]. The drawback of these techniques is

that data in cache blocks are lost when they are turned off and the extra power needed to access interconnection network and L_2 cache (if L_1 miss occurs) or main memory (if L_2 miss occurs) to reload data may negate any power saving and may degrade performance. To avoid these pitfalls, we put cache blocks into drowsy mode [8].

A drowsy cell exploits dynamic voltage scaling to reduce leakage power. Each cache block can switch between high and low (drowsy) supply voltages. When a cache block is idle its voltage is set to low supply voltage. Due to short-channel effects in deep-submicron processes, leakage current reduces significantly in idle cache blocks. The combined effect of reduced leakage current and voltage results in a dramatic saving in static power. Whenever an SM sends a request to a cache, the cache controller checks the condition of the voltage of the cache line. If the accessed line is in normal mode, no extra delay is incurred, because the power mode of the line can be checked concurrently with the read and comparison of the tag. However, if the line is in drowsy mode, we need to prevent the discharge of the bit-line of the cache line because it may read out the wrong data. We need to wait an extra cycle to switch the supply voltage back to normal mode before reading out the data.

One implication of drowsy cell is that execution time of programs may increase since drowsy cells require extra time to wake-up. We use a two-level scheduler [17] to select a warp for execution. Each cycle, the scheduler selects a ready warp from the active list and sends it for execution. To hide wake-up latency of drowsy cells, the scheduler should send the source operands of a load/store instruction to the memory unit before the associated instruction is issued. To handle this, the scheduler can issue a warp and concurrently look into active warps to find the warp that is going to be issued in the next cycle. Thus, one can eliminate the overhead of drowsy cells with 1-cyle wake-up delay. Similarly, the scheduler can look into active list and send information of the warp to the memory unit *n* cycles ahead and may wake-up drowsy cells to hide *n* cycles of wake-up delay. So, the two-level scheduler is able to hide the latency of drowsy cells. However, we also evaluate a scheduler which is not able to check the warps ahead of time. In Section 5, we explore the performance impact of drowsy cells with different wake-up latencies assuming that it is not feasible to hide the latency of drowsy cells.

4.2 Reducing Dynamic Power Using Active Mask

In the previous section, we used drowsy cells to reduce leakage power when a cache block is idle. However, when the cache block is accessed all bytes within the block are placed in ON state. For example, in Fermi family, each cache block is 128-byte. So, when SM executes a load/store instruction, the whole 128-byte is woken up. Accessing such a large number of SRAM cells incurs significant dynamic power because of activating word-lines, bit-lines, and sense amplifiers.

As shown in Table 1, the percentage of active threads varies across the benchmarks. Because of branch divergence, some warps cannot fill the whole 32 slots. However, in existing implementations of GPGPUs, a warp with partial utilization still activates the whole cache block. This means that we have to pre-charge word-line (WL), bit-line (BL), bit-line-bar (BLB), and sense amplifiers for the whole cache

block although a subset of the cache block is used for warp execution. One way to reduce dynamic power of the cache blocks is to access only portions of cache blocks that are accessed by active threads. GPUs use an active mask which indicates active threads within a warp. The mask is a vector of 32 bits and each bit corresponds to a thread. When a branch instruction diverges, the bits corresponding to active threads are set and the rest are cleared. Hence, we can use active mask to disable portions of cache blocks associated with inactive threads.

We use the Divided Word Line (DWL) [23] technique to implement active mask aware access to caches. Figure 5 illustrates the structure of DWL. In DWL, the WL is segmented into several Small WLs (SWLs). Each SWL enables or disables accessing to the portion of cache block attached to it. For our work, each SWL covers 4-byte of the cache block. The output of a row decoder is connected to SWLs. GPU architecture is suited for easy integration of DWL into caches. A warp's active mask has all information required to determine which SWL should be active or inactive. Each SWL is activated by an AND gate which has two inputs, the horizontal line coming from the row decoder and the vertical line coming from the active mask. DWL reduces dynamic power since whenever a cache block is accessed those bytes within the cache block that correspond to the inactive threads are disabled.

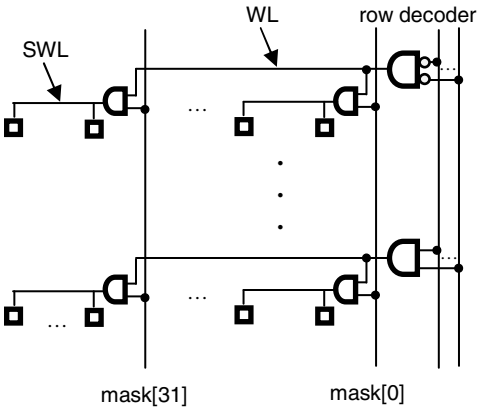


Fig. 5. Structure of DWL

5 Methodology and Results

We used GPGPU-Sim (version 3.1.1) [3] to evaluate our power aware optimization techniques. GPGPU-Sim is a publicly available, detailed cycle-based simulator for GPGPUs. We configure the simulator to closely match NVIDIA's Fermi GTX480 as recommended in the GPGPU-Sim manual (Table 2). We use a collection of benchmarks from CUDA SDK [18], Rodinia Benchmark suite [20], and Parboil Benchmark suite [21] (Table 1). We ran the benchmarks until completion or for 1 billion instructions, whichever comes first.

5.1 Experimental Results

In this section, we report power saving in L_1 and L_2 caches. Figure 6 shows static, dynamic, and total power saving in L_1 and L_2 caches. For each benchmark, the first bar represents static power in caches with drowsy mode relative to the static power of the baseline scheme. Bars less than one show power reduction. In order to quantify the leakage current in caches, we modeled a cache based on 6T SRAM cells in HSPICE. We used the technology files from Predictive Technology Models (PTM) [12] with feature size of 32-nm and nominal voltage of 0.9V. We found that the state of SRAM cells can be maintained if V_{dd} is reduced up to 0.2V. While an ideal drowsy cell can work at 0.2v, in practice it is necessary to add safety margin to take into account noise and also mismatch between transistors. Table 3 shows static power for nominal voltage and reduced voltages in a row of L1 and L2 caches. Even when V_{dd} is reduced to 0.4v, the static power is less than 8% of static power when cache cells operate at full V_{dd}. For the rest of this section, we assume drowsy cells operate at 0.4v.

Table 2. GPGPU-Sim Configuration

Number of SMs	16
Warps/Shader	48
Threads per warp	32
PEs/SM	32
Registers per core	32768
$L_1 (size/assoc/line)	16KB/4-way/128B
$L_2 (size/assoc/line)	768KB/16-way/128B
Memory controller	FR-FCFS

Table 3. Static power in a row of $L_1/$L_2

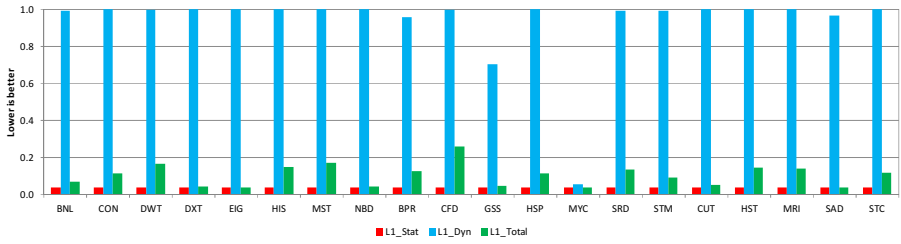
V_{dd}(v)	0.2	0.3	0.4	0.9
Static Power $L_1/$L_2(mw)	0.04/0.26	0.056/0.36	0.08/0.53	1.081/6.7

The second column in Figure 6 shows dynamic power in L_1 and L_2 caches with active mask relative to dynamic power of the baseline scheme. We extracted resistance and capacitance of SWLs based on the model used in CACTI v6.0 [15]. Similar to static power, we used HSPICE with PTM [12] and feature size of 32-nm to estimate dynamic power. The dynamic power depends on warp utilization of the benchmarks (Table 1). Benchmarks with low warp utilization, i.e. MYC, show significant dynamic power saving. On the other side, benchmarks such as CUT that usually have 32 active threads do not benefit from this technique. Benchmarks with moderate warp utilization, i.e. GSS, have limited dynamic power saving. On average, the dynamic power reduces by 7% and 24% in L_1 and L_2 caches, respectively.

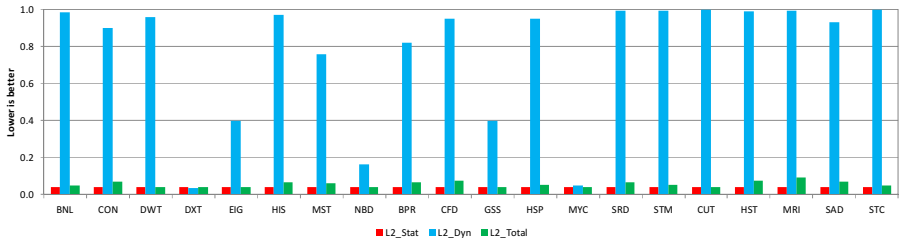
The third column in Figure 6 shows the relative total power saving. The combined system first changes the state of a requested cache block from drowsy to active. Then, based on active mask, it decides which portion of the cache block should be activated. Benchmarks with low warp utilization, i.e. MYC, have the highest power saving because they take advantage of both the leakage and the dynamic power saving techniques. On average, the total power is reduced by 90% and 96% in L_1 and L_2 caches, respectively.

As we discussed in Section 4.1, a two level scheduler can activate a cache block ahead of time and avoid any penalty due to wake-up delay. However, if it is not

feasible to hide wake-up latency (for example if GPGPU uses a scheduler other than the two level scheduler), we assume that this delays execution of the warps. To quantify the effect of wake-up latency, we ran the benchmarks with one and two extra cycles overhead. Note that these latencies are in addition to the latency of the baseline cache. Figure 7 shows performance of a GPGPU with drowsy cache relative to the baseline scheme. Bars less than one show slow-down. A GPGPU has many warps and if a warp is stalled due to cache delay, the GPGPU can issue and execute another warp. Hence, the performance changes slightly with wake-up delay. On average, the performance of the benchmarks changes by less than 0.3% when wake-up delay is one and two cycles. In some benchmarks, i.e. STM, execution time reduces when wake-up delay increase. We analyzed these benchmarks and found that the sequence of executed warps changes with wake-up delay. In the new sequence, cache miss rate reduces and this improves performance of these benchmarks slightly.



a)



b)

Fig. 6. Static, dynamic, and total power saving in a) L_1 and b) L_2 caches

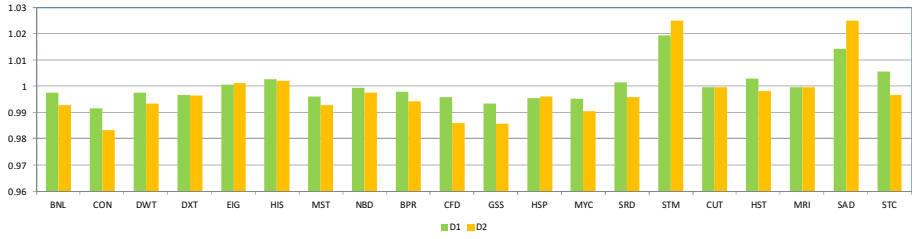


Fig. 7. Performance impact of drowsy cache with one and two cycles wake-up delay

6 Related Work

Gebhart et al. [2] proposed a unified local memory which can dynamically change the capacity of register, shared memory, and cache on a per application basis. Existing implementations of GPGPUs use a one-size-fit-all policy and hard-partition local storage of an SM in design time. However, GPGPU applications have diverse local storage requirements and a single memory unit is often most critical to performance of a given application. Gebhart et al. [2] proposed a unified memory architecture that aggregates different memory units and allows a flexible allocation based on applications' requirements. The tuning that this flexibility enables improves both performance and energy of GPGPUs.

Sankaranarayanan et al. [7] proposed adding tinyCache to reduce power of L_1 data cache. A tinyCache is a small filter inserted between a PE and an L_1 data cache and intercepts accesses to the shared L_1 cache. The main challenge of tinyCache is cache coherency. Since each PE has a private tinyCache, it is necessary to maintain coherency across tinyCaches of an SM. To reduce coherence overhead, Sankaranarayanan et al. proposed to either evict content of tinyCache into L_1 cache (e.g. for barriers) or bypass tinyCache (e.g. for atomic operations). TinyCache is able to reduce power of L_1 data cache by filtering out a sizable portion of memory accesses to the L_1 cache.

The above techniques can be used along with our optimization techniques to reduce power consumption of caches in GPGPUs further.

Warped register file [16] uses compiler to turn off unallocated registers and places the rest into drowsy mode to reduce leakage power. It also avoids charging bit-lines and word-lines of registers associated with inactive threads to reduce dynamic power. Our work is different from warped register file since we focus on the power of caches in GPGPUs.

This paper is an extension of our previous work [19] on L_1 data caches in GPGPUs. We have studies inter-access cycle and warp utilization in L_2 cache and found that the behavior of L_2 cache is similar to L_1 cache. We applied drowsy cell and active mask to the cache blocks and reduced static, dynamic, and total power of L_1 and L_2 caches.

7 Conclusion

This paper proposes two power-aware optimization techniques that target static and dynamic power of L_1 and L_2 caches in GPGPUs. Due to large inter-access distance of cache blocks, GPGPUs provide unique opportunities to reduce power. Our first optimization technique puts cache blocks into drowsy state and brings them to active state only when they are accessed. Given the large pool of warps in GPGPUs, this aggressive drowsy state management technique impacts performance negligibly. The second technique exploits active masks and eliminates activation of unused portions of cache blocks. These two optimization techniques combined are able to reduce power of L_1 and L_2 caches by 90% and 96%, respectively.

Acknowledgment. This work was supported by the National Sciences and Engineering Research Council of Canada.

References

1. Kaxiras, S., Hu, Z., Martonosi, M.: Cache decay: Exploiting generational behavior to reduce cache leakage power. In: Proceedings of ISCA, pp. 240–251 (2001)
2. Gebhart, M., et al.: Unifying primary cache, scratch, and register file memories in a throughput processor. In: Proceedings of MICRO-45, pp. 96–106 (2012)
3. Bakhoda, A., Yuan, G., Fung, W., Wong, H., Aamodt, T.: Analyzing CUDA workloads using a detailed GPU simulator. In: Proceedings of ISPASS (April 2009)
4. Bakhoda, A., Kim, J., Aamodt, T.: Throughput-effective On-chip Networks for Manycore Accelerators. In: MICRO (2010)
5. Fung, W., Sham, I., Yuan, G., Aamodt, T.: DynamicWarp Formation and Scheduling for Efficient GPU Control Flow. In: MICRO (2007)
6. Boettcher, M., et al.: MALEC: A Multiple Access Low Energy Cache. In: Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 368–373 (2013)
7. Sankaranarayanan, A., Ardestani, E.K., Briz, J.L., Renau, J.: An Energy Efficient GPGPU Memory Hierarchy with Tiny Incoherent Caches. In: ISLPED, pp. 9–14 (2013)
8. Flautner, K., et al.: Drowsy caches: Simple techniques for reducing leakage power. In: Proceedings of ISCA, pp. 148–157 (2002)
9. NVIDIA Corp. NVIDIA's Next Generation CUDA Compute Architecture: Fermi (2009)
10. NVIDIA. CUDA Programming Guide Version 5.0 (2013)
11. NVIDIA Corp. NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110 (2012)
12. Arizona state university predictive technology model, `http://ptm.asu.edu`
13. Demers, E.: Evolution of AMD graphics, AMD Fusion Developer Summit (2011)
14. Agrawal, A., Jain, P., Ansari, A., Torrellas, J.: Refrint: Intelligent refresh to minimize power in on-chip multiprocessor cache hierarchies. In: Proceedings of HPCA (2013)
15. Muralimanoharet, N., et al.: Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0. In: Proceedings of MICRO (2007)
16. Abdel-Majeed, M., Annavaram, M.: Warped Register File: A Power Efficient Register File for GPGPUs. In: Proceedings of HPCA (2013)
17. Gebhart, M., et al.: Energy-efficient mechanisms for managing thread context in throughput processors. In: Proceedings of the ISCA, pp. 235–246 (2011)
18. NVIDIA. CUDA C/C++ SDK code samples (2013)
19. Atoofian, E.: Reducing Static and Dynamic Power of L1 Data Caches in GPGPUs. In: Proceedings of HPPAC, Phoenix AZ (2014)
20. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J., Lee, S.-H., Skadron, K.: Rodinia: A Benchmark Suite for Heterogeneous Computing. In: IISWC (2009)
21. Stratton, J.A., et al.: Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing (2012)
22. Zhou, H., et al.: Adaptive mode-control: A static-power-efficient cache design. In: Proceedings of International Conference on Parallel Architectures and Compilation Techniques (2001)
23. Yoshimoto, M., et al.: A divided word-line structure in the static ram and its application to a 64k full cmos ram. IEEE Journal of Solid-State Circuits 18(5), 479–485 (1983)