LoSCache: Leveraging Locality Similarity to Build Energy-Efficient GPU L2 Cache

Jingweijia Tan[1], Kaige Yan[2], Shuaiwen Leon Song[3] and Xin Fu[4]

[1]College of Computer Science and Technology, Jilin University, Changchun, China
[2]College of Communication Engineering, Jilin University, Changchun, China
[3]HPC Group, Pacific Northwest National Laboratory, Richland, USA
[4]ECE Department, University of Houston, Houston, USA
Emails: {jtan, yankaige}@jlu.edu.cn, shuaiwen.song@pnnl.gov, xfu8@central.uh.edu

***Abstract*—This paper presents a novel energy-efficient cache design for massively parallel, throughput-oriented architectures like GPUs. Unlike L1 data cache on modern GPUs, L2 cache shared by all the streaming multiprocessors is not the primary performance bottleneck but it does consume a large amount of chip energy. We observe that L2 cache is significantly under-utilized by spending 95.6% of the time storing useless data. If such "dead time" on L2 is identified and reduced, L2's energy efficiency can be drastically improved. Fortunately, we discover that the SIMT programming model of GPUs provides a unique feature among threads: instruction-level data locality similarity, which can be used to accurately predict the data re-reference counts at L2 cache block level. We propose a simple design that leverages this *Lo*cality *S*imilarity to build an energy-efficient GPU L2 *Cache*, named *LoSCache*. Specifically, LoSCache uses the data locality information from a small group of CTAs to dynamically predict the L2-level data re-reference counts of the remaining CTAs. After that, specific L2 cache lines can be powered off if they are predicted to be "dead" after certain accesses. Experimental results on a wide range of applications demonstrate that our proposed design can significantly reduce the L2 cache energy by an average of 64% with only 0.5% performance loss.**

***Index Terms*—GPU, cache, energy-efficiency, locality similarity**

I. Introduction

With the high computational throughput, modern GPUs are extensively applied for accelerating high performance computing (HPC) applications. On-chip caches are adopted to enhance data locality of GPU applications. However, recent studies [1]–[4] have suggested that the cache hierarchy is not efficiently utilized in current GPUs. These works primarily focus on addressing the inefficiency of L1 cache due to its performance-criticality. In contrast, with a much larger capacity and lower placement in the memory hierarchy, the GPU L2 cache is largely ignored in the previous works since it is not a primary performance bottleneck. However, as a large on-chip SRAM structure [5], inefficient utilization of L2 cache can significantly increase the overall GPU energy consumption.

To better understand L2 cache efficiency, we first analyze its utilization. Figure 1 illustrates the dead-time percentage of L2 cache lines across different applications (Section III). We refer to the time duration during which a cache line stores useless data as "dead time". The percentage value is calculated as $Total\_Dead\_Time/(Execution\_Time \times Num\_Cache\_Lines)$. The figure clearly shows that on average GPU L2 cache lines spend 95.6% of the execution time being dead. Among all the benchmarks, only MM has relatively lower (i.e., $< 60\%$) cache line dead-time. This indicates that the current GPU L2 cache is poorly utilized and consumes a significant portion of energy in storing data that will not be re-referenced again in the future.

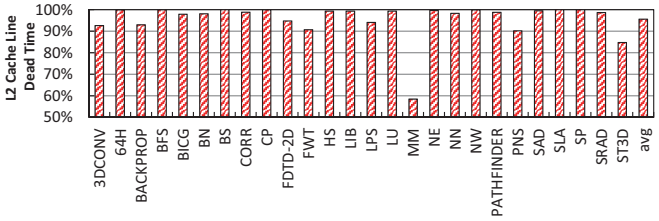

Fig. 1. L2 cache line dead time on baseline architecture.

To reduce the unnecessary energy waste caused by L2 underutilization, we need to identify and significantly reduce these dead-time periods. In this paper, we make a key observation: data locality in GPU applications has instruction-level similarity among threads due to the unique feature of the SIMT (*Single Instruction, Multiple Thread*) programming model. This finding can be used to accurately predict the data re-reference counts at L2 cache line level. With this information, L2 cache lines can be powered off during the "dead-time" periods. Based on this idea, we propose to leverage the *Lo*cality *S*imilarity to build an energy-efficient GPU L2 *Cache* (*LoSCache*) to reduce the unnecessary energy waste in L2 caused by its utilization inefficiency. Based on the locality similarity, it uses the information from a small group of CTAs to accurately predict the L2-level data re-reference counts of the remaining CTAs. Such locality prediction is conducted at data level instead of cache line level. After this, LoSCache powers off the cache lines that are predicted to be dead after certain accesses to conserve energy. Experimental results demontrate that LoSCache can reduce the L2 cache energy by an average of 64% with only 0.5% performance degradation due to its high prediction accuracy.

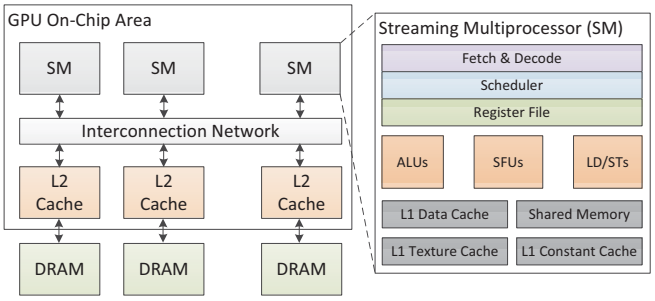

Fig. 2. Baseline GPU architecture.

II. Background: GPU Architecture

Figure 2 shows the architecture of a modern GPU[1]. The GPU on-chip area is composed of a number of in-order streaming multiprocessors (SM), multiple distributed L2 cache banks in different memory partitions [1], [2], [7], and interconnection network (ICNT) that handles the communication among SMs and memory partitions. The right part of the figure shows a zoom-in view of an SM. It is composed of a scalar front-end that fetches and decodes the same instruction of a group of threads, and an SIMD back-end that executes the threads with different operands.

GPU uses a single-instruction multiple-thread (SIMT) programming model, which means threads within the same program kernel execute the same instruction with different thread operands. A kernel is composed of a grid of threads; a grid is divided into a set of *Cooperative Thread Arrays* (CTAs), or *thread blocks*. Each CTA is composed of hundreds of threads. Threads are distributed to SMs at the granularity of CTAs, and threads within a single CTA communicate via the shared memory and synchronize at a barrier if desired. The number of CTAs that execute concurrently in the same SM depends on their per-CTA resource requirements. Usually, not all the CTAs from a program kernel can be allocated to the SMs in a single round. Thus the remaining CTAs wait until their previous CTAs in the SMs finish and release their resources. In the SM pipeline, a number of threads from the same CTA are grouped together, called a *warp*. Threads within the same warp execute in SIMD mode and the warp scheduler selects the ready warps for execution.

The caches in GPU are organized hierarchically, composed of SM-private L1 caches and a shared L2 cache. The L1 caches are only accessed by threads from the same SM, while the L2 cache is accessed by all the threads of the kernel. The L1 data cache is typically write-evict and write no-allocate [2], [7], while the L2 cache is write-back with write-allocate [2], [7]. To simplify the design, GPU caches are non-inclusive non-exclusive without hardware coherence [2], [7].

III. Instruction-Level Data Locality Similarity

Since GPU uses a SIMT programming model, all the threads within the same kernel execute the same program code with different operands. Thus, when instructions with the same program counter (PC) are executed by different threads, they tend to exhibit similar behaviors. As a result, the data access requests generated by the same-PC instructions from different threads tend to have similar locality behavior as well. We explain the reason by analyzing the memory access instructions from two representative benchmarks, `LIB` and `MM`. By analyzing the data accesses, we observe that all accesses for `LIB` are intra-CTA locality, while more than 99% of the accesses for `MM` are inter-CTA locality. Figure 3 and 4 further illustrate two examples of data accesses from these two applications. For `LIB` shown in Figure 3, every thread loads data from the global memory to its own shared memory space (line 9) based on the CTA ID and thread ID (line 3). Since the CTA ID and thread ID are both one dimensional in `LIB`, the data accesses have intra-CTA locality. Thus, even though different data is accessed by different threads when executing the same-PC instructions in this code, the data tends to show similar locality behavior.

```
1  __global__ void Pathcalc_Portfolio_KernelGPU2(float *d_v)
2  {
3    const int    tid = blockDim.x * blockIdx.x + threadIdx.x;
4    const int threadN = blockDim.x * gridDim.x;
5    int   i, path;
6    float L[NN], z[NN];
7    for(path = tid; path < NPATH; path += threadN){
8      ...
9      d_v[path] = portfolio(L);
10     ...
11   }
12 }
```

Fig. 3. Code example of `LIB` with intra-CTA locality.

```
1  __global__ void matrixMul( float* C, float* A, float* B, int wA, int wB)
2  {
3    int bx = blockIdx.x;
4    int by = blockIdx.y;
5    int tx = threadIdx.x;
6    int ty = threadIdx.y;
7    ...
8    int aBegin = wA * BLOCK_SIZE * by;
9    int bBegin = BLOCK_SIZE * bx;
10   ...
11   for (int a = aBegin, b = bBegin; a <= aEnd; a += aStep, b += bStep) {
12     ...
13     AS(ty, tx) = A[a + wA * ty + tx];
14     BS(ty, tx) = B[b + wB * ty + tx];
15     ...
16   }
17   ...
18 }
```

Fig. 4. Code example of `MM` with inter-CTA locality.

Figure 4 shows the code example from `MM`. In this case, every thread loads the data from the global memory (i.e., A and B) to their own shared memory space (i.e., AS and BS) (line 13 and 14) only based on the thread ID and one dimension of the block id (i.e., blockIdx.x or blockIdx.y) (line 8 and 9). When executing the same instruction, threads from different CTAs access the same data from the global memory as long as their thread IDs and one dimension of their CTA ids are the same, resulting in inter-CTA locality. In addition, re-reference count in L2 for shared data is related to the number of CTAs that have the same dimension value. *"Related" but not "equal" is*

[1]In this paper, we focus on the NVIDIA CUDA [6] terminology to describe the GPU architecture and the programming model. Note that our work is independent of the terminology itself.

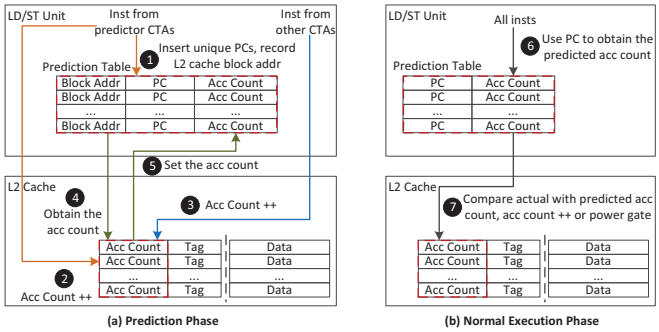

Fig. 5. Architecture extensions and implementation of LoSCache. - - - - locates the minor hardware extensions.

because there are cases where CTAs with the same dimension value locate on the same SM, which does not belong to inter-CTA locality.

As shown in these examples, data locality in GPU applications has unique instruction-level similarity among different threads. We will leverage this opportunity to predict the data re-reference count next.

IV. Design Methodology For LoSCache

In this section, we propose a mechanism that leverages the instruction-level data-locality similarity among SIMT threads to build an energy-efficient GPU L2 cache. First, We describe the prediction process for data re-reference counts. We then demonstrate how to leverage this prediction to design an energy-efficient GPU L2 cache, named *LoSCache*. Finally, we analyze the overhead of LoSCache. The goal of LoSCache is to gain the maximum energy savings from the "dead time" of L2 cache lines.

A. LoSCache: Structure

In order to implement LoSCache, we add minor hardware extensions to the LD/ST unit and L2 cache of the baseline GPU architecture, shown as the red-highlighted components in Figure 5. We discuss the structures of these two hardware extensions as follows.

Prediction Table: Shown in Figure 5, a prediction table is placed in the LD/ST unit, which is aimed to predict the re-reference counts of data at L2 cache line level based on the instruction-level data-locality similarity discussed previously. Some control logic are also added together with the prediction table. Each entry in the prediction table contains the locality prediction information about L2 cache accesses from instructions with an unique PC. Figure 5(a) shows the details of a prediction table entry, including the *block address* of the L2 cache line, the *PC* of the instructions that access this L2 cache line, and the predicted *access count* to this cache line. The *PC* and the *access count* are used to indicate the predicted locality while the *block address* is used to locate the corresponding L2 cache line at the end of the prediction phase (Section IV-B).

L2 Cache Extension: In order to assist the locality detection, we also extend the tag store of the L2 cache, shown in Figure 5. Each cache line is extended with an *access count* field to indicate the actual access count of the current data in this cache line. When the current data is evicted, the access count field is reset to 0. This field is used to update the locality information of the prediction table at the end of the prediction phase, and find the dead time of the corresponding cache line for energy savings during the normal execution. For power gating, a sleep transistor is added to each cache line to control the power supply, and switching between on and off mode.

B. LoSCache: Operation

The three major operation phases of LoSCache design are as follows:

Predictor Selection. During the program execution, different threads tend to execute in a similar pace on GPU due to the SIMT model and hardware scheduling. In addition, the access interval to the same memory address is usually long because of the massive thread interleaving. Thus it is very difficult to predict the data locality of each memory request under the current scheduling policies (e.g., Greedy-Then-Oldest [1]). Our basic idea to address this issue is to select some "predictors" and execute them faster in order to make runtime decisions for the others. A simple way to implement this on GPU is to detect the locality information of all memory accesses in a single thread, and use it to predict the locality for all the other threads within the same kernel. However, an accurate prediction will not be reached until the predicting thread completes a large amount of the memory access instructions. Also, the faster execution of this thread will be impeded by the CTA-level barriers. Additionally, the locality information of a single thread may be biased. In this work, we propose an efficient mechanism to cut down the drawbacks from single thread prediction. We randomly select one CTA from each SM as the *predictor*. This is because the characteristics of certain CTAs may be dramatically different from the others in the same application. For example, in one kernel of `BFS`, certain threads from a CTA are assigned with more workloads than the others, resulting in the imbalance of memory accesses among threads. Using the requests from one specific CTA to predict the other CTAs will be inaccurate. Our random predictor selection reduces the possibility of selecting monotonic CTAs as the predictors. Choosing multiple CTAs across SMs as predictors further reduces prediction bias as well.

Prediction Phase. After the predictors are selected, the program goes through the *prediction phase*. During this phase, threads within the predictors have higher execution priority. Meanwhile, threads from the other CTAs execute in a lower priority; their ready threads will not execute until none of the threads in the predictors are ready (e.g., stalls due to data dependencies). In this way, the predictors can generate the locality information in advance for prediction. Figure 5(a) shows the prediction phase. When memory access instructions from the predictor CTAs are issued, their PCs and the L2 cache block addresses are inserted into the prediction table in parallel with the memory access request generation (1). The prediction table of each SM only stores entries with unique

PCs. So before a PC is inserted, it will be first compared with the PCs of the existing entries in the same table. If there is a match, this PC will not be inserted. Meanwhile, the memory access requests are generated in LD/ST units and sent to the L2 cache. The corresponding access count of the L2 cache line is increased by one from both the predictor CTAs (2) and the other normal CTAs (3). At the end of the prediction phase, the access count information of each entry in the prediction table is updated: using the block address of each entry to locate the corresponding L2 cache line (4), and then updating the access count of the prediction table using the access count of the corresponding L2 cache line (5).

Normal Execution. Once the prediction phase is completed, the information in the prediction table becomes available for future use. We refer to this phase as the normal execution phase, shown in Figure 5(b). During this phase, all the memory request instructions check the prediction table using their instruction PCs to obtain the predicted access count of the corresponding L2 cache line (6). This is in parallel with the address calculation in the LD/ST unit. When accessing to the targeted data, the predicted access count is then compared with the current access count of the L2 cache line (7). If the current access count is smaller than the prediction, the current access count is increased by one. Otherwise, it means the access is the last one according to the prediction. The corresponding L2 cache line is then powered off after this access in order to save energy. If the predicted access count is one, the accessed data will not be stored in the L2 cache since by prediction it will not be referenced in L2. Note that all the L2 cache lines are powered off at the beginning of the program and woke up when first accessed. This is because some applications are observed to only use a small percentage of L2 cache lines during the entire execution.

C. Optimizations

Setting Prediction Period. The prediction period is critical to the accuracy and effectiveness of the prediction. The most naive way is to terminate the prediction period when all the predictor CTAs finish their execution. However, for applications that have a large amount of L2 accesses, this greatly reduces the effectiveness of the prediction since a decent prediction could very likely be available earlier than that time, limiting the energy-saving potential. Thus we also consider L2 cache access intensity when determining prediction period. Based on the experiments, we set 100 L2 cache accesses as the threshold. So the prediction period is finished after 100 L2 cache accesses or all the predictor CTAs are completed, whichever comes first.

Increasing Prediction Accuracy. If the predicted re-reference count is smaller than the actual re-reference count, the cache line is powered off earlier (i.e., early gating). This case will increase the cache miss rate and hurt the performance. We apply an adaptive mechanism against this scenario. After a cache line is power-gated, the tag still stores the information before the next data fills in. If the same data hits the tag of this cache line, it indicates that the actual re-reference count of the data is larger than the predicted. We then add a threshold for the future prediction of the accesses from this PC. In this case, the re-reference count for the future memory accesses is equal to the predicted value plus the threshold. Every inaccurate prediction will increase the threshold by one. Based on our experiments, a maximum threshold of three covers almost all the applications we studied. Therefore, we set its upper bound to be three.

D. Overhead Analysis

Area Overhead. The prediction table size is related to the number of unique PCs for the L2 cache accesses during the prediction phase. Our experiment indicates the largest number of unique PCs is 21 among all benchmarks (in `NW`), which is used as the number of entries. For each entry, we set 25 bits for the L2 block address, 32 bits for the PC field, and 6 bits for the predicted access count (i.e., the highest L2 access count among all applications is 32). Therefore, we estimate the size of the prediction table as 21*(25+32+6)=165 Bytes per LD/ST unit. We set the access count in the L2 cache line to 6 bits, same as that in the prediction table. Thus the storage overhead is 6 bits per L2 cache line. We use CACTI 6.5 [8] to calculate the area overhead of the added storage structures under 40nm technology. For the control and gating logic, we apply a similar method used in [9], [10] to aggressively estimate their area overhead as three times of the added storage structures. We estimated the total area overhead of the design is $0.449mm^2$, only 0.084% of the entire GPU chip area.

Latency Overhead. We assume accessing the prediction table takes 1 additional cycle. However, this latency can be well hidden since it occurs in parallel with memory address generation in the LD/ST units. Additionally, there are also wake-up periods for the power-gated cache lines. Similarly, the wake-up process does not negatively affect the overall performance since it occurs in parallel with the data access to the lower level memory hierarchy. In other words, the cache-line reservation periods require to fetch new data from the memory during cache misses anyway.

V. Experimental Methodology

Simulation Environment: Our proposed LoSCache is evaluated using GPGPU-Sim V3.2.2 [11], a widely-adopted cycle-accurate simulator for GPU architecture research. The configuration parameters of the baseline GPU architecture are set as follows: there are 15 SMs with 32 SIMD width and 1.4GHz, the warp scheduling policy is Greedy then oldest, each SM contains 16KB L1 cache with 128B line and 4-way associativity; the unified L2 cache is 768KB with 128B line and 16-way associativity, the L2 cache is partitioned into 6 banks with 128KB per bank; the DRAM includes 6 channels, each channel contains a 16-entry queue and the scheduling policy is out-of-order first ready first come first serve. We use a modified GPUWattch [5] to evaluate the dynamic and leakage energy consumption of LoSCache. We also include the energy consumption of accessing our hardware extensions

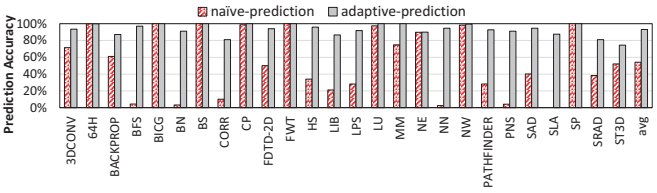

Fig. 6. Prediction accuracy comparison between naive- and adaptive-prediction.

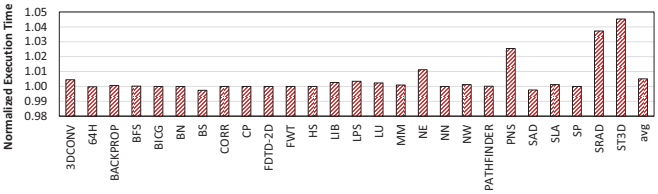

Fig. 7. Normalized execution time of LoSCache against the baseline architecture.

in our evaluation (i.e., accessing the prediction table in LD/ST unit and extended L2 tag).

Benchmarks: In order to faithfully evaluate our proposed technique, we collect a large set of 26 representative applications from the NVIDIA CUDA SDK [12], PolyBench [13], Rodinia benchmark suite [14] and Parboil benchmark suite [15], with default inputs. In our experiments, all workloads run to completion on the simulator.

VI. Experimental Results And Analysis

A. Prediction Accuracy

We first evaluate the prediction accuracy of LoSCache. Figure 6 shows the prediction accuracy for naive- and adaptive-prediction (introduced in Section IV-C to increase prediction accuracy), respectively. The prediction accuracy indicates the percentage of correct predictions (i.e., predicted access count is the same as the actual access count) among all the predictions. The figure shows that the prediction accuracy for the naive-prediction is low, which is 54% on average. In contrast, the adaptive prediction greatly increases the prediction accuracy to 93% on average. Based on the accuracy evaluation standard, we check if the prediction is absolutely accurate for each data. For some cases, the naive-prediction can only predict a similar locality but not the exact. However, by only adaptively increasing a small threshold, the prediction accuracy is significantly improved. For example, adding the threshold increases the prediction accuracy of `BFS` from 4% to 97%. On the other hand, the adaptive prediction is unable to further increase the prediction accuracy for some benchmarks, including `64H`, `BICG`, `BS`, `CP`, `FWT`, `NE`, `SP`, since their prediction accuracy is already very high with the naive-prediction. From this point forward, we use adaptive prediction in LoSCache evaluation.

B. Performance Overhead

There are two potential reasons for LoSCache to incur performance overhead: (1) the slight ICNT traffic increase at the end of the prediction phase, which may degrade the performance; and (2) cache lines suffer from the early power-gating

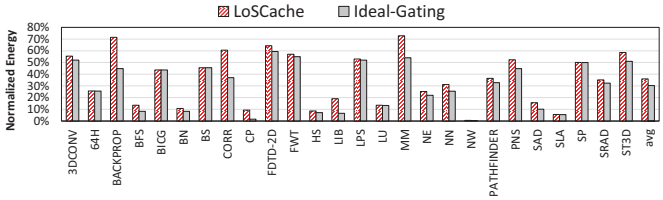

Fig. 8. Normalized energy consumption for LoSCache and Ideal-Gating against the baseline L2.

due to inaccurate prediction, which increases L2 miss rate. However, Figure 7 demonstrates that LoSCache design does not noticeably affect the overall performance. On average, the execution time only increases by 0.5%. This result is consistent with the high prediction accuracy from our LoSCache design, shown in Figure 6. Also, although the prediction accuracy for some benchmarks is lower, such as `ST3D` and `SRAD`, it only slightly impacts the performance. For example, the execution time of `ST3D` is only increased by 4% due to the small amount of early gating through incorrect prediction. Additionally, even though `NW` has a 14% ICNT traffic increase, its performance degradation is less than 1%. This is because its baseline ICNT traffic is so small that the traffic increase does not play a critical role in determining the overall performance.

C. Energy Consumption

We also evaluate the energy consumption under LoSCache. Figure 8 shows the normalized energy results against the baseline L2. As it shows, LoSCache greatly reduces the energy consumption from the baseline design. On average, the energy consumption is only 36% of the baseline scenario. LoSCache is effective for most benchmarks due to the low data re-reference counts in the L2 cache and its accurate prediction. For example, the energy consumption of `NW` is less than 1% of the baseline L2 cache due to its extremely low L2 cache utilization. On the other hand, LoSCache can only reduce the energy consumption of `MM` by 27% because of its high data re-reference counts. We also compare LoSCache with the Ideal-Gating technique, which is an oracle case that all the L2 cache lines are powered off immediately once they expire. On average, Ideal-Gating further reduces the L2 energy consumption by 6% compared to LoSCache. The reasons are twofold. Although the prediction accuracy of LoSCache is high, it is not 100%. Thus it wastes a small portion of energy on storing the expired cache lines due to the inaccurate prediction. The other reason is that LoSCache cannot power gate any expired cache line during the prediction phase. We further evaluate the energy saving of the entire GPU chip. Based on our evaluation, the energy consumption of GPU L2 cache is around 8% of the total GPU chip energy. As a consequence, LoSCache reduces 6.2% of the total GPU chip energy on average.

VII. Related Work

Due to the massive thread-level parallelism and the limited capacity, the GPU caches are not efficiently used. Several recent studies have addressed this problem to mitigate the

performance loss due to cache conflicts or resource contention [1]–[4]. For example, Rogers et al. [1] proposed a cache-conscious wavefront scheduling to maintain the intra-CTA cache locality in L1 cache. Xie et al. [3] proposed to improve the cache performance via a coordinated static code analysis and dynamic cache bypassing. These techniques all effectively improve the L1 cache performance by exploiting data locality and avoiding thrashing. However, the L2 cache utilization in these designs remains low. In contrast, our proposed LoSCache technique addresses the inefficiency of the L2 cache in GPU by leveraging the instruction-level data-locality similarity to reduce unnecessary energy waste. LoSCache is compatible with all these L1 cache management techniques and can achieve further energy savings.

Several previous works were also proposed to reduce the energy consumption of CPU caches [16]–[19]. For example, Kaxiras et al. proposed to power off a cache line if it is not accessed after a long period of time [16]. For this technique, the prediction for cache line dead-time is coarse-grained and only based on the cache line itself. In contrast, our technique leverages the unique locality similarity of GPU programs to accurately predict the dead-time in advance and powers off a cache line immediately after the last request of a data access. Additionally, several other works have been proposed to reduce the energy consumption of GPU caches. For example, Wang et al. proposed to put the L1 and L2 caches to sleep mode when they are not active to save energy [20]. This work aims to reduce energy when the entire cache is not active. In contrast, our LoSCache is able to reduce the L2 energy when it is active by predicting the cache-line dead-time.

VIII. Conclusions

L2 cache is underutilized in modern GPUs. It stores useless data for the majority of time, as a result, data in L2 has no or few future re-references. To address this inefficient utilization of L2 cache, we propose LoSCache, a simple and effective energy-efficient GPU L2 cache design. LoSCache leverages GPU's unique instruction-level data-locality similarity caused by the SIMT programming model to predict the data re-reference counts in L2. Based on this prediction mechanism, LosCache powers off the L2 cache lines if they are predicted to be dead after certain accesses. Experimental results show that LoSCache dramatically reduces the energy consumption of the GPU L2 cache with negligible performance loss.

Acknowledgment

The work is supported in part by National Natural Science Foundation of China (NSFC) under Grant No. 61802143, Jilin Scientific and Technological Development Program under Grant No. 20180101046JC, and Research Project by the Education Department of Jilin Province under Grant No. JJKH20190159KJ. This research is also partially supported by U.S. National Science Foundation grants CCF-1619243, CCF-1537085(CAREER), CCF-1537062, and DOE CENATE project.

References

[1] T. G. Rogers, M. OConnor, and T. M. Aamodt, "Cache-conscious wavefront scheduling," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec 2012, pp. 72–83.

[2] X. Chen, L. Chang, C. I. Rodrigues, J. Lv, Z. Wang, and W. Hwu, "Adaptive cache management for energy-efficient gpu computing," in *International Symposium on Microarchitecture*, Dec 2014, pp. 343–355.

[3] X. Xie, Y. Liang, Y. Wang, G. Sun, and T. Wang, "Coordinated static and dynamic cache bypassing for gpus," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2015, pp. 76–88.

[4] D. Li, M. Rhu, D. R. Johnson, M. O'Connor, M. Erez, D. Burger, D. S. Fussell, and S. W. Redder, "Priority-based cache allocation in throughput processors," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2015, pp. 89–100.

[5] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "Gpuwattch: Enabling energy optimizations in gpgpus," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY, USA: ACM, 2013, pp. 487–498.

[6] NVIDIA, "CUDA C Programming Guide," http://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[7] I. Singh, A. Shriraman, W. W. L. Fung, M. O'Connor, and T. M. Aamodt, "Cache coherence for gpu architectures," in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2013, pp. 578–590.

[8] S. J. E. Wilton and N. P. Jouppi, "Cacti: an enhanced cache access and cycle time model," *IEEE Journal of Solid-State Circuits*, vol. 31, no. 5, pp. 677–688, May 1996.

[9] T. G. Rogers, D. R. Johnson, M. O'Connor, and S. W. Keckler, "A variable warp size architecture," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, June 2015, pp. 489–501.

[10] J. Tan, S. L. Song, K. Yan, X. Fu, A. Marquez, and D. Kerbyson, "Combating the reliability challenge of gpu register file at low supply voltage," in *2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)*, Sept 2016, pp. 3–15.

[11] 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*, April 2009, pp. 163–174.

[12] "Nvidia cuda sdk," https://developer.nvidia.com/cuda-downloads.

[13] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a high-level language targeted to gpu codes," in *2012 Innovative Parallel Computing (InPar)*, May 2012, pp. 1–10.

[14] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, Oct 2009, pp. 44–54.

[15] "Parboil benchmark suite," https://github.com/abduld/Parboil.

[16] S. Kaxiras, Z. Hu, and M. Martonosi, "Cache decay: exploiting generational behavior to reduce cache leakage power," in *Proceedings 28th Annual International Symposium on Computer Architecture*, June 2001, pp. 240–251.

[17] H. Liu, M. Ferdman, J. Huh, and D. Burger, "Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency," in *2008 41st IEEE/ACM International Symposium on Microarchitecture*, Nov 2008, pp. 222–233.

[18] Z. Hu, S. Kaxiras, and M. Martonosi, "Timekeeping techniques for predicting and optimizing memory behavior," in *2003 IEEE International Solid-State Circuits Conference, 2003. Digest of Technical Papers. ISSCC.*, Feb 2003, pp. 166–485 vol.1.

[19] A.-C. Lai, C. Fide, and B. Falsafi, "Dead-block prediction amp;amp; dead-block correlating prefetchers," in *Proceedings 28th Annual International Symposium on Computer Architecture*, June 2001, pp. 144–154.

[20] Y. Wang, S. Roy, and N. Ranganathan, "Run-time power-gating in caches of gpus for leakage energy savings," in *2012 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2012, pp. 300–303.