Adaptive GPU Cache Bypassing

Aishwarya Lekshmi Chithra
chithra@wisc.edu.com
Electrical and Computer Engineering
University of Wisconsin-Madison

Rajesh Shashi Kumar
rshashikumar@wisc.edu
Electrical and Computer Engineering
University of Wisconsin-Madison

December 21, 2021

Abstract

CPU caches leverage the temporal and spatial locality properties of data to reduce the memory access time thereby increasing the performance and reducing the power consumption of a system. GPUs on the other hand, make use of a huge number of parallel threads to hide the long memory latency. Due to the fundamental difference in the multi-threaded throughput-oriented execution, GPU cache structures are inefficient in performance and energy consumption for general purpose workloads. Traditionally, GPUs were focused on graphic workloads. But the massive increase in parallel performance over the recent years has made GPUs useful in accelerating compute intensive GPGPU workloads. These workloads however, exhibit large data sets which are sometimes streaming in nature. The low temporal locality and large reuse distances in these applications exacerbate the problems with cache efficiency in GPUs. In this project, we propose an adaptive GPU fine-grained adaptive cache bypassing scheme in hardware to improve the efficiency of L1-D caches both in terms of power and performance.

The proposed cache bypassing technique tries to dynamically bypass the blocks that are less likely to be accessed again, helping to keep reusable data in the cache longer. This technique helps save energy in two ways: (1) by reducing the execution time of the program by increasing the hit rate (2) by reducing the number of unnecessary insertions and deletions from the cache.

1 Introduction

Memory is central to Von-Neumann architecture and has been a limiting factor in recent times to exploit improvements in compute performance fully. Approaching the memory wall has necessitated specialization in computing through workload-specific profiling and optimizations for reduced memory access times. Parallel computing has found widespread use in High Performance Computing (HPC) for scientific computing and Data Science. Data parallelism and increase in memory accesses have made determining the right cache policies a challenging problem, especially in GPUs. Recently, there has been a lot of traction towards using GPGPU programming to run common compute intensive applications at high speed. The small caches shared between multiple threads in GPU architecture are not efficient in extracting high performance with the streaming applications in GPGPUs. Such applications are observed to have a high reuse distance, and hence most reusable data gets evicted from the cache

to be replaced by zero-reuse blocks (blocks that are evicted from cache before they are touched again). This results in cache pollution, reducing the hit rates, and wasting power in saving the zero-reuse blocks.

2 Background

2.1 Motivation

GPUs rely on thread level parallelism to hide the latency associated with memory accesses. When one thread results in a miss in the cache and the data needs to be fetched from the memory, a huge number of threads are available that can be run instead. This massive multi-threading approach makes it difficult to use data locality effectively. Hence, GPUs rely less on traditional caches compared to CPUs and more on programmable scratchpad and texture caches which requires a great deal of programming effort to use. Such fine-grained cache management techniques with compiler hints also run the problem of ensuring code compatibility with changing device and memory configurations. A hardware-controlled cache eliminates the programming effort, solves the compatibility issue and makes it easier to be used. But due to the long reuse distance of data in GPGPU applications, the useful data gets replaced in the cache by streaming data before it can be used. Increasing the cache size can potentially fix this problem, but it is not a desirable solution in GPUs as adding more compute units in the available area can provide significantly more performance improvement. So, there is a need to develop an adaptive cache management technique that can improve the efficiency of small caches by not storing zero-reuse blocks.

2.2 Bypass Predictor

Cache bypassing is a widely used technique in CPUs to reduce thrashing. It selectively bypasses some memory accesses so that some other accesses can still reside in the cache and eliminate early eviction of the latter. Our project implements a dynamic cache bypass prediction technique for GPGPUs called Adaptive GPU Cache Bypassing (AGCB). The implemented bypass predictor dynamically identifies and bypasses zero-reuse blocks to the compute units, thereby reducing cache pollution and increasing the lifetime of the reusable blocks in the cache.

Many CPU bypass predictors in the literature use memory access addresses to predict a zero-reuse block. But GPGPUs employ Single Instruction Multiple Data (SIMD) methodology in its threads resulting in a huge number of different data addresses but the same Program Counter (PC) for the instruction. It requires large storage to save the adrress related history for the predictor but only requires small storage to store PC related history[8]. Since area savings is critical in GPGPUs, the latter approach is preferred in the current implementation.

The PC-based bypass predictor stores the history of reuse for a particular PC in a prediction table, and this table is looked up to decide whether or not to bypass a block or not. If the predictor mispredicts the block as reusable and to be placed in the cache, energy is wasted in a pointless insertion and the eviction of maybe reusable data. But if the predictor mispredicts the block to be bypassed, the block will never be placed in L1 cache and,

hence, irreversible. Therefore, a verification step is needed where an additional field called *bypassBit* is stored in L2 cache. When a prediction is made to bypass a block, the *bypassBit* for the corresponding L2 entry is extracted to be compared with the current decision. If the *bypassBit* indicates that the block was previously bypassed, indicating a misprediction, the block will not be bypassed and will be placed in the L1 cache.

3 Related Work

Jia *et al.*[5] proposed Memory Request Prioritization Buffer (MRPB) [1], which employs cache bypassing to ease the programming effort and increase the GPU performance. To reduce the contention of multiple threads on the cache, the MRPB attempts to reorder the requests in such a way that requests from related threads are sent to the cache at the same time to preserve access locality. This paper employs cache bypassing when there is a contention on the hardware resources associated like cache line, miss queue, MSHR.However, this form of bypassing is not adaptive or intelligent enough to factor in the temporal locality of the bypassed block leading to reduction in the performance. On the other hand, AGCB attempts to classify the blocks as zero-reuse or reusable and has a verification mechanism with an L2 cache to not bypass reusable blocks.

Rogers *et al.*[10] proposed Cache-Conscious Wavefront Scheduling (CCWS) which changes the access pattern to reduce thrashing in the L1 cache by limiting the number of threads that can access the cache at a particular time. Instead of predicting the reference interval of blocks, this method changes the reference interval to avoid thrashing. However, AGCB does not at any point limit the number of threads accessing the cache, thereby better hiding the memory access latencies. Further, the CCWS technique can be tuned to trade-off power with performance whereas the AGCB improves power and performance in tandem.

Zheng *et al.*[12] implemented Adaptive Cache and Concurrency Allocation(CCA) where the memory access patterns are used to predict locality and the data locality is maximized by limiting the number of warps that are cached. In contrast, the warps that have less locality is bypassed from the L1 cache. This design also limits the number of threads accessing the cache, unlike AGCB. The CCA design uses a footprint table and an access pattern table to predict the cached and bypassed warps, while AGCB is a simpler design using only a single smaller table called the prediction table.

Khan *et al.*[8] introduced Sampling Dead Block Prediction for Last-Level Caches in CPUs which served as a foundation for the work on AGCB. Instead of using a large number of accesses and evictions to build a history for the predictor, SDBP uses PC to predict the reusability of a particular block, thereby ensuring that the predictors require less storage. However, SDBP uses a set of PCs to build the history of the predictor while AGCB uses the last PC for the same. The predictor in SDBP is used for the last level cache(LLC) whereas the one for AGCB is used with the L1 cache but they both make use of the fact that they have insufficient temporal locality to leverage for their corresponding cache. In CPUs the higher levels of caches would have stripped the blocks of its temporal

locality whereas in GPGPU workloads the large reuse distance between data gives incomplete information regarding the temporal locality.

Around the same time, Chen *et al.*[3] came up with Adaptive Cache Management for Energy-efficient GPU Computing where along with bypassing, the focus is also on reducing the congestion of on-chip resources like MSHR. The massive amount of multithreading employed in GPUs can result in congestion of resources, unlike in CPUs and this results in stalls which decreases the performance further. Opportunities for bypass and the possibility of congestion are computed at runtime. The bypass policy is coordinated with the warp throttling to exploit the data locality and reduce the pressure on on-chip resources. This design limits the number of threads accessing the cache unlike in AGCB, resulting in underutilization of available threads. The design is more complex and requires more storage than AGCB but it can reduce the resource congestion.

The priority-based cache allocation (PCAL) by Li *et al.*[9] proposes a lightweight thread throttling technique that classifies threads according to their priority and provides preference in the cache for the high priority threads to reduce resource contention. Furthermore, the scheme classifies the threads as regular and non-polluting threads and only the regular threads have access to the cache while the non-polluting caches still run concurrently. Even though this is not exactly a cache bypassing technique, it achieves almost the same result through thread throttling while also not sacrificing the number of threads that run simultaneously and the utilization efficiency associated with that.

Scratchpad and Texture memories are used along with compiler hints to improve the reuse of data in Kandemir *et al.*.[6] However, these software solutions will not work for every application, and there's a significant amount of programming effort to tailor them to each application.

4 Architecture

The PC-based bypass predictor consists of a 256-entry prediction table parallel to the L1 cache as shown in Figure 1. The prediction table is an array of 4-bit saturating counters indexed using the hashed value of the PC. L1D cache stores the hashed value of the last PC to access every block as an additional field called *hashedPC* in the tag array. The hash function masks the upper bits and extracts the last 8bits of the PC to generate *hashedPC*. Since GPGPU applications have a very small number of distinct PCs using the last 8 bits to index the prediction table would not result in a collision. LRU is used as the replacement policy in this design. To predict whether a block has reuse, the counter value indexed by the *hashedPC* is accessed. If this counter value is greater than a predefined threshold value, the block is predicted as zero-reuse and is bypassed. Else, the block is brought into the cache by replacing a victim block. The counter in the corresponding entry of the victim block in the prediction table is updated to indicate less reuse of the block. The prediction table saturating counters are initialized to 15, and a threshold value of 8 is chosen for this design to have an aggressive bypassing technique by default and caching only when the

counter value has decremented to below the threshold value. This strategy is adopted since GPGPU workloads are streaming in nature and has less reuse.

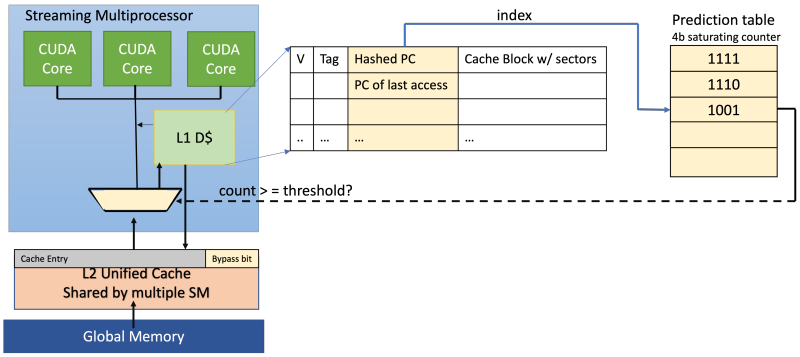


Figure 1: Structure of the PC-based bypass predictor

Figure 2 shows the flowchart of the bypass predictor. For every read access, the tag array of the L1D cache is searched for the tag of the address of access. If there is a match, it implies a HIT condition and that the previous PC to access the block resulted in a reused block. Therefore, it is necessary to update the prediction table to indicate reuse for the previous PC. Hence, the counter corresponding to the prediction table entry for the previous *hashedPC* is decremented. The *hashedPC* field in the tag array is further updated to save the hashed value of the new PC and the LRU status of the block is also updated to indicate the current access. If the tag array comparison indicates a MISS condition, it is checked whether the counter value corresponding to the current PC is greater than or equal to the threshold value. If the check is positive, that block is predicted to be zero-reuse and it is decided to be bypassed, pending additional verification. If it is negative, the block is predicted to be reused and it is decided to bring the block into the cache from L2 and a request is sent to L2. If the prediction is to bring the block into L1D cache, a victim is chosen to be evicted if applicable, according to the replacement policy of the cache. The prediction table counter for the victim *hashedPC* is incremented to indicate that the corresponding PC might result in a zero-reuse block. The bypass decision is shared with the L2 cache along with the request for the data and is stored in L2 as a *bypassBit* for future verification.

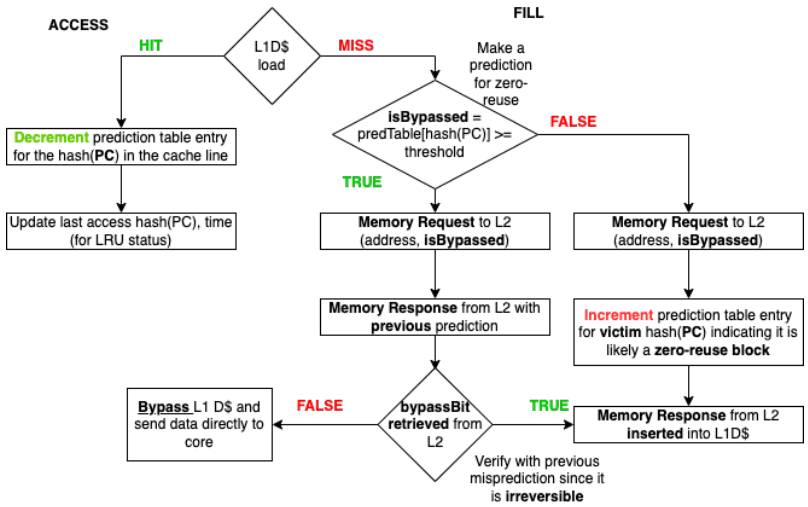


Figure 2: Flow chart for the bypass predictor

To avoid any irrecoverable misprediction consequences, when a block is predicted to be bypassed, the *bypassBit* stored in the L2 cache is checked. If the bit is set, it means that previously this PC was mispredicted to be zero-reuse and bypassed. Upon detecting the misprediction, the bypass decision from the predictor is overridden and the block is brought into cache. The *bypassBit* in L2 is set back to 0 to indicate reuse for future verification. On the other hand, while checking if the *bypassBit* is found to be not set, it indicates that the prediction is correct and the block is bypassed to the compute unit.

5 Experimental Methodology

5.1 Simulation environment

We implement the design of the proposed adaptive cache bypassing technique on GPGPU-sim. GPGPU-Sim provides a detailed simulation model of Nvidia GPU architectures [1]. GPU cache organization exhibits significant changes with evolution in device architectures. GPGPU-sim is an open-source simulator that includes support for updated models across different architectures, including recent offerings from Nvidia. The architectures of interest for the evaluation in this work are Volta (QV100) and Fermi (GTX480).

In our initial benchmarking simulations to obtain baseline metrics on GPGPU-sim, we wrote scripts to launch runs and collect metrics. We found this to be a tedious approach for repeated experiments with varying configurations. In an effort to make experiments portable and reproducible, we later incorporated the use of Accel-Sim. Accel-Sim is a simulation framework that allows simulation and validation of GPU models [7]. For this project, we only leverage Accel-Sim as a front-end to GPGPU-sim to run simulations. We found this framework to be useful in improving research productivity.

Workload	Application domain	CUDA kernels in program	Maximum L1-D access count in each kernel
b+tree	Search	2	4.2M
backprop	Pattern recoginition	2	1.5M
bfs	Graph Algorithms	60	20M
dwt2d	Discrete wavelet transforms in Video Compression	20	1.7M
gaussian	Linear Algebra	1086	9.4M
hotspot	Physics simulation	2	0.8M
lud	Linear Algebra	140	1.4M
nn	K-nearest neighbours in Data mining	1	26K

Table 1: Description and characteristics of benchmarks chosen from Rodinia

5.2 Benchmarks

As described earlier, our intuition is that adaptive cache bypassing in hardware should significantly reduce programmer effort involved in fine-grained cache management to accelerate GPGPU-workloads. We chose 8 benchmarks shown in Table 1 from the Rodinia suite for evaluation. The benchmarks were chosen to be specifically representative of GPGPU workloads across different application domains to validate our hypothesis about programmer effort. These benchmarks were obtained from the gpu-app-collection [2] that has been integrated with Accel-Sim. The default input sizes were used in all simulations. The number of CUDA kernels in each workload has been indicated to show the variation in parallel SM utilization since the bypass prediction logic is implemented per SM. By determining the maximum L1-D cache access count across kernels generated in each of these benchmarks, we concluded that *bfs* is the most memory-intensive workload in this selection. We also included *backprop*, which has been classified as a cache insensitive workload in previous work [4]. In addition to these benchmarks, we wrote a simple matrix multiplication workload with variable matrix size to incrementally test our implementation changes in GPGPU-sim for both functionality and performance variation.

6 Implementation

6.1 Differences from the reference design

Before we discuss the implementation, we highlight the key differences in the simulation environment and design decisions between the implementation in our project and the reference [11] that this work is based on. The differences are presented in Table 2. The reasons for some of these differences are as follows:

- Nvidia GPUs include support for cache bypassing in software through the use of compiler hints. Not only does this indicate a potential value in bypassing mechanisms, but this would be later useful in comparing the efficacy of our work on adaptive cache bypassing in hardware with existing methods. Evaluation on the Fermi GPU model was only used in obtaining baseline metrics since it is the closest available approximation to the GPUs that were available at the time when the reference was published.

- We chose a different simulator since GPGPU-sim is widely used and includes support for recent GPU architecture models

	Reference[11]	Implementation in this project
GPU model used	AMD GCN	Nvidia Volta (QV100) Fermi (GTX480)
Simulator	In-house based on gem5	GPGPU-sim
Baseline L1-D cache configuration	8-way 16KB, 64B blocks	64-way, 32KB, 128B blocks
Baseline L2 cache configuration	16-way 256KB, 64B blocks	24-way, 6MB, 128B blocks
Threshold value for bypassing	Not specified	8
Hash function for PC	Not specified	Modulo
Initialization value of prediction table	Not specified	15

Table 2: Differences from reference work

- The differences in baseline cache configuration are mainly due to the difference in device architecture. We intentionally used the configurations in the unmodified form as it would aid in comparing baseline metrics (on newer architectures) with our evaluation. We did not see any analysis value in replicating the cache configuration that the reference [11] used since we use a fundamentally different GPU model.
- The threshold, hash function and initialization value of prediction table were not specified in the reference [11]. The reasons for choosing these specific values have been described in the Architecture section.

6.2 Implementation of adaptive cache bypassing in GPGPU-sim

We implement the structures needed to implement bypassing in a series of steps. The modular approach allowed us to verify the correctness of each block in our implementation since the project involves dealing with shared structures such as the prediction history table that is updated on several events. The implementation code changes discussed in this section have been made available in this repository (github.com/rajesh-s/cs752-adaptive-gpu-cache-bypassing). We discuss the system configuration specific details such as cache sizes and organization under the evaluation section.

Developing an understanding of the functioning, cache interactions and code structure of GPGPU-sim was an important first step in implementing this work. In this section, we describe a simulator-focused overview of the implementation changes for adaptive cache bypassing.

GPGPU-sim has a cache inheritance structure that allows reuse of common functions such as tag array lookup, for example, between the various caches as shown in Figure 3. Figure 1 illustrates the top-level summary of changes. We now describe implementation details for each subsection below.

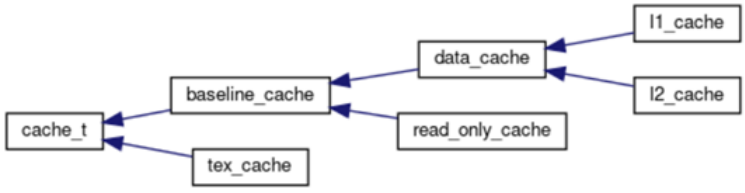


Figure 3: Inheritance diagram of cache classes in GPGPU-sim[1]

6.2.1 Hashed PC storage in L1-D cache

The tag array comprises of all the lines (banks x associativity x sets) in the L1-D cache created to be each of the type *cache_block_t*. The tag array is instantiated in *class baseline_cache* (in *gpu-cache.h*) for every instantiation of *class l1_cache* (in *shader.cc*). The *cache_block_t* structure primary stores the tag and block address corresponding to each line in the cache. Further, there are several specific fields (eg: last access time) depending on whether the architecture uses a line cache or a sector cache. We added the storage for hashed PC corresponding to the last PC that accessed a specific line in the *cache_block_t* structure as shown in Figure 4. This field is updated whenever a new line is allocated or a fill occurs using the *cache_block::allocate()* and *cache_block::fill()* methods respectively.

```
104   struct cache_block_t {
105     cache_block_t() {
106       m_tag = 0;
107       m_block_addr = 0;
108       m_hashed_pc = 0; // Rajesh cs752 hashed PC for L1
```

Figure 4: Adding storage in each cache line for hashed PC value in *gpu-cache.h*

6.2.2 Prediction table

We implement the prediction table as a member of *class l1_cache* (in *gpu-cache.h*). This is because we want each SM to have a dedicated prediction table that stores local history based on the memory access patterns of the kernel running on the specific SM. The prediction table entries are implemented as a 4bit saturating counter. We use the simple technique of modular hashing to index into the table while keeping the overheads to a bare minimum. The hashing is based on the lowest 8bits of the Program Counter value that initiated the load access. Correspondingly, there are 256 entries in the prediction table. A pointer to the shared prediction table data structure is passed to other functions for threshold computation and increment or decrement operations.

```
1745  class l1_cache : public data_cache {
1746   public:
1747    l1_cache(const char *name, cache_config &config, int core_id, int type_id,
1748             mem_fetch_interface *memport, mem_fetch_allocator *mfcreator,
1749             enum mem_fetch_status status, class gpgpu_sim *gpu)
1750        : data_cache(name, config, core_id, type_id, memport, mfcreator, status,
1751                     L1_WR_ALLOC_R, L1_WRBK_ACC, gpu) {}
1752
1753    virtual ~l1_cache() {}
1754
1755    uint8_t l1d_prediction_table[256]; //  L1D Prediction table CS752
```

Figure 5: Creation of the L1-D prediction table in *gpu-cache.h*

6.2.3 Decrementing the prediction table entry

The lower the value of an entry in the prediction table, the lower is the likelihood of bypassing the load access at a particular PC. Thus, we decrement the value of prediction table entries when a HIT occurs for a block in L1D ,

indicating likely reuse in upcoming accesses. L1D cache accesses are initiated in the *ldst_unit::cycle()*. The accesses that result in a L1D HIT follow the function call flow shown in Figure 6. Thus, we implement the decrement logic in the overloaded function for L1-D hits called *data_cache::rd_hit_base_l1d*.

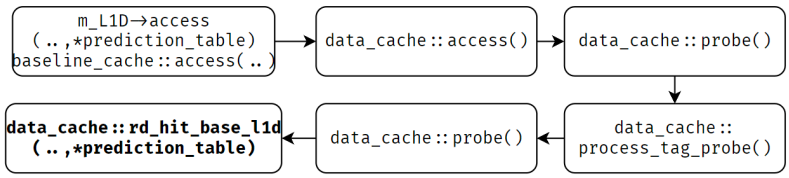


Figure 6: Function call flow in case of L1-D HIT

```
1897  enum cache_request_status data_cache::rd_hit_base_l1d(
1898      new_addr_type addr, unsigned cache_index, mem_fetch *mf, unsigned time,
1899      std::list<cache_event> &events, enum cache_request_status status, uint8_t *l1d_prediction_table) {
1900    new_addr_type block_addr = m_config.block_addr(addr);
1901    // fprintf(stdout, "Data Cache Normal PC %d\n",mf->get_pc()); Verified
1902    uint8_t storedhashedPC = m_tag_array->get_hashed_pc_from_tag(addr, mf); // Rajesh CS752
1903    if(l1d_prediction_table[storedhashedPC] > 0 ){ // Saturating counter stays 0 on 0
1904      l1d_prediction_table[storedhashedPC]--;
```

Figure 7: Decrementing the prediction table entry in *gpu-cache.cc*

6.2.4 Incrementing the prediction table entry

The entries of prediction table are desired to reflect a 4bit saturating counter. However, the lowest data type available in C++ is *uint8_t* corresponding to 1byte. So, we use the *uint8_t* data type with bound checking on increment so that the value does not exceed 15.

As described in the pseudo-code, the prediction table entry is incremented when a L1D MISS causes a cache block to be evicted. We assume an ON_FILL policy on read misses in our implementation. This means that in case of a MISS, the cache line allocation happens when the response is received from lower levels of memory corresponding to the missed line. When such an insertion results in an eviction, we increment the prediction table entry for the hashed PC of the victim block.

The L1D fill() method in the *ldst_unit::cycle()* is responsible for handling responses received from L2. We tracked the control flow of this function illustrated below. Logic inside *tag_array::probe()* determines whether a block was evicted. We capture it in the victim valid pointer in deciding whether the increment to prediction table entry should take place.

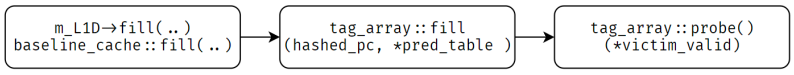


Figure 8: Control flow when responses are received from L2

```
643     enum cache_request_status status = probe(addr, idx, mask, victim_valid, false, NULL);
644     //fprintf(stdout,"AISH ON_FILL, %s, %d victim_valid %d\n",__func__, __LINE__, victim_valid);
645     // assert(status==MISS||status==SECTOR_MISS); // MSHR should have prevented
646     // redundant memory request
647     bool isBypassed = false;
648     int threshold = 8; // From SDBP paper
649     if(l1d_prediction_table[get_hashed_pc_from_tag(addr,NULL)] >= threshold){
650       isBypassed = true;
651     }
652
653     if(l1d_prediction_table[get_hashed_pc_from_tag(addr,NULL)] < 15 && victim_valid && isBypassed==false)
654      {
655       l1d_prediction_table[get_hashed_pc_from_tag(addr,NULL)]++ ;// Rajesh CS752 Victim Hashed PC
```

Figure 9: Incrementing the prediction table entry in *gpu-cache.cc*

6.2.5 Bypass bit storage in L2 cache

The L2 cache shared across SMs on Nvidia GPUs are non-inclusive non-exclusive caches, meaning cache lines that are brought into the L1 cache are also brought into the L2 cache, but an L2 line is evicted silently without recalling L1 cache [4]. We leverage this property of GPU cache organization to implement the verification logic to avoid mispredictions outlined in the Architecture section. For this, we include a *bypassBit* in every cache line entry in the tag_array for L2 cache. This is similar in implementation to the storage for hashed PC described in section 6.2.1. This bit stores the bypass bit received from L1 on a cache block request to L2 that results in a HIT at the L2 cache. It is used as historical prediction data as described in pseudo-code to avoid irreversible bypassing.

6.2.6 Storage and retrieval of bypass information between L1 and L2 for verification

We now implement a mechanism to send the bypassing decision determined through threshold computation in L1 to the block in the L2 cache on a miss. This is necessary to implement the verification of the prediction decision as discussed in section 4. For this, we use the *mem_fetch* object in GPGPU-sim. The *mem_fetch* object carries information related to memory requests and responses through the interconnect and caches. The *baseline_cache::send_read_request()* method in *gpu-cache.cc* is used by L1 to initiate accesses to L2 on miss by framing a *mem_fetch* object. We embed the bypass decision computed in L1 within a newly created field of *mem_fetch* object. If the access results in a HIT in the L2 cache, it returns the previously stored value of bypass decision to L1 along with the response to block request. The bypass bit received in the request packet is then stored in L2 for verification during the next access. The code snippet that implements this logic is included in Figure 10

```
521         if (status == HIT) {
522           if (!write_sent) {
523             // L2 cache replies
524             assert(!read_sent);
525             if (mf->get_access_type() == L1_WRBK_ACC) {
526               m_request_tracker.erase(mf);
527               delete mf;
528             } else {
529               bool isBypassed = false;
530               if(mf->get_sentfroml1() == 777){
531                 isBypassed = mf->get_isBypassed(); // This will be used to later update L2
532                 //752fprintf(stdout,"The prediction stored to L2: %d Addr:%u \n",isBypassed, mf->get_addr());
533                 bool bypassBit = m_L2cache->get_bypass_bit_from_l2(mf->get_addr(), mf); // Read existing bypass bit from L2
534                 //752fprintf(stdout,"The prediction retrieved from L2: %d Addr:%u \n",bypassBit, mf->get_addr());
535                 mf->set_isBypassed(bypassBit); // Sending old bypassBit back to L1
536               }
537               mf->set_reply();
538               mf->set_status(IN_PARTITION_L2_TO_ICNT_QUEUE,
539                              m_gpu->gpu_sim_cycle + m_gpu->gpu_tot_sim_cycle);
540               m_L2_icnt_queue->push(mf); // Pushing read payload from L2 to interconnect
541               if(mf->get_sentfroml1() == 777){
542                 m_L2cache->set_bypass_bit_from_l2(mf->get_addr(), mf, isBypassed);
543               }
544             }
545             m_icnt_L2_queue->pop();
```

Figure 10: Implementing the bypass bit storage and retrieval in *l2cache.cc*

6.3 Verification of submodule implementation

We implemented our design as individual submodules as described in section 6.2. To ensure that each of these components carry out the desired function, we verified the following operations to be working as expected. The verification was carried out using *fprintf* statements and a matrix multiply workload with variable input sizes.

- Hashed PC is updated on L1 cache for every L1 read access.
- Prediction table updates occur on hits and misses for line/sector cache.
- L1-D and L2 cache are able to exchange bypass bit history for verification.
- Threshold computation and bypassing mechanism occur when L1-D cache acccesses are initiated.

6.4 Integration

With the individual submodules implemented, the bypassing mechanism is now introduced by integrating them. The focus of our work is on global memory access loads. Bypassing is performed in two situations:

- Accesses to L1-D cache lookup are bypassed based on the threshold computation. This is implemented in the function *ldst_unit::memory_cycle()* illustrated in Figure 11.
- New cache block insertions as a result of memory responses received from L2. The bypassing here determines whether the line is directly forwarded to the compute unit or inserted into cache. We assume an on fill allocation policy for read misses in our project. This is implemented as illustrated in Figure 12

```
2052    if (CACHE_GLOBAL == inst.cache_op || (m_L1D == NULL)) {
2053      bypassL1D = true;
2054    } else if (inst.space.is_global()) {  // global memory access
2055      // skip L1 cache  if the option is enabled
2056      if (m_core->get_config()->gmem_skip_L1D && (CACHE_L1 != inst.cache_op)) bypassL1D = true;
2057      if (m_L1D->l1d_prediction_table[(uint8_t) inst.pc] >= 8 && access.get_type() == GLOBAL_ACC_R) {bypassL1D = true;
```

Figure 11: Implementing the bypass logic to avoid cache lookup in *shader.cc*

```
2602          bool bypassL1D = false; uint8_t temp_pc = 0;
2603          address_type currPC = mf->get_pc();
2604          temp_pc = (currPC == -1) ? (uint8_t) mf->get_original_mf()->get_pc() : (uint8_t) currPC;
2605          if (CACHE_GLOBAL == mf->get_inst().cache_op || (m_L1D == NULL)) {
2606            bypassL1D = true;
2607          } else if (mf->get_access_type() == GLOBAL_ACC_R ||
2608                     mf->get_access_type() ==
2609                         GLOBAL_ACC_W) {  // global memory access
2610            if (m_core->get_config()->gmem_skip_L1D) bypassL1D = true;
2611            if (m_L1D->l1d_prediction_table[temp_pc] >= 8 && mf->get_access_type() == GLOBAL_ACC_R) bypassL1D = true; // Rajesh CS752  threshold
2612               fprintf(stdout,"Bypassed here\n"); mf->print(stdout);
```

Figure 12: Implementing the bypass logic to avoid cache block insertion in *shader.cc*

6.4.1 Debug

With the above integration in place, we saw that the performance metrics did not vary in evaluation in comparison with the baseline. We spent a significant portion of our time on debugging the issues in integration. The confidence through the verification of individual submodules was found to be helpful in reducing the number of unknowns during debug. The following observations were made in the process:

- By executing the benchmarks with varying cache configurations and tracking values of prediction table entries, we concluded that performance differences were not observed primarily due to the strategy we used to initialize the prediction table entries. Using a value of 0 (not bypass) for initializing prediction table entries turned out to be a pessimistic approach given the workloads that were being used in evaluation. Instead, using the value of 15 for initialization ensures that cache blocks are bypassed by default and only when a reuse pattern is noted (through decrements of the counter), should the block be inserted in the cache. This aggressive strategy for bypassing is probably better suited for streaming workloads.

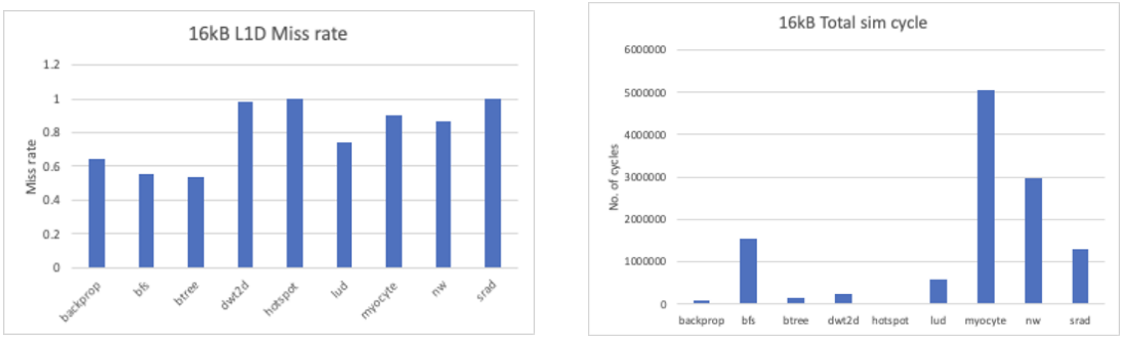
- Setting the initialization value to 15 in the previous step, uncovered that we were not bypassing as many accesses as we expected. Further debug revealed that implementing our design with a on miss policy can be tricky given that the allocation of cache block happens when a miss is logged in the MSHR as opposed to when a cacheline insertion is about to take place. For this reason, we found it useful to assumed a on fill approach, where allocation happens when the response from L2 is received on a L1 cache block miss.

- With the initialization value and allocation policy figured out, we stumbled upon two strange issues. First, a segmentation fault where some mem_fetch objects traversing through the interconnect between the caches seem to have been dropped. We guessed that this was most likely due to the way sector accesses are handled in GPGPU-sim, that we did not previously account for. The second issue that we encountered, was a deadlock issue where the simulator could not make progress. We are using *gdb* to debug this issue and have not been

able to fully address it as we experienced some difficulty in tracking the large number of memory requests in progress. We suspect that this could be due to a mismatch in number of memory requests and responses upon bypassing.

7 Evaluation

To understand the effect of using different GPU model, we tried to reproduce the results from [11] for the same cache configuration. Experiments were conducted on GPUGPU-sim without the bypass logic on Rodinia-2.0 benchmark to obtain the stats as shown in Figure 13. Metrics of miss rate and the number of simulation cycles taken by the applications are considered for our evaluation. Bypassing technique is expected to reduce the miss rate and decrease the total number of simulation cycles.



(a) Miss rate

(b) Number of simulation cycles

Figure 13: Simulation results of Baseline

The above results showed that there was little value to replicating the cache configuration used in [11]. We decided to use a baseline that was closer to modern architectures. For this reason, we used the system configuration indicated in Table 3 as our baseline for future evaluation.

Simulator	Accel-sim/GPGPU-sim
Architecture	Turing (QV100)
L1 cache	Sector cache, 128B blocks, 64-way, LRU replacement, On fill allocation policy, 32KB
L2 cache	Sector cache, 128B blocks, 24-way, 32 sets, 6MB capacity

Table 3: System configuration used in evaluation

The performance metric used to determine the efficient use of cache is hit ratio. This should be indicative of reuse of blocks and reduced cache pollution. We measured the baseline values for hit ratio using the rodinia-3.1 benchmark suite. Figure 14. The high hit ratio indicates the frequency of L1-access in these workloads which may serve as a candidate for bypassing.

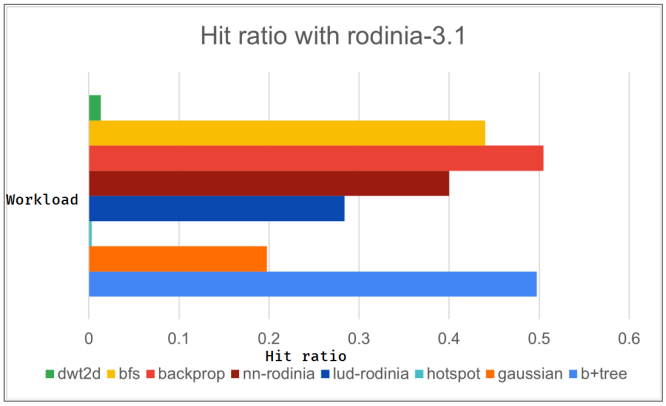
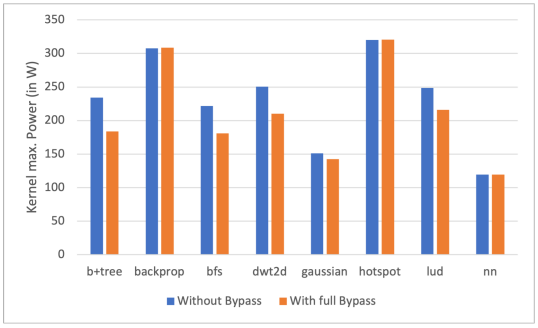
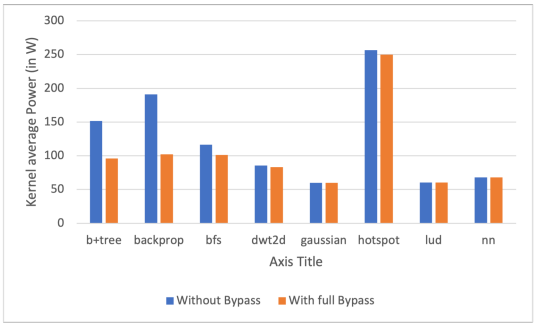


Figure 14: Hit ratio observed on rodinia-3.1

Though we were not able to get the metrics we expected with our design operating at expected level due to the issues outlined in section 6.4.1, we wanted to investigate the possible benefits of bypassing in terms of power efficiency. For this, we used full bypassing of L1 cache through a GPGPUsim switch to observe the energy savings. Our intuition is this would be the closest approximation to an inefficient bypass predictor. The results shown in Figure 15 were obtained using Accel-Watch on the Volta architecture. As seen in both plots of average and maximum power measured across the execution of all kernels, there is significant power savings observed due to bypassing on memory intensive workloads such as *bfs*, *backprop* and *btree*.



(a) Maximum Power



(b) Average Power

Figure 15: Kernel Power results for no bypassing and full bypassing of L1D cache

8 Contribution

Through this project we have attempted to develop an in-depth understanding of the Accel-sim and GPGPU-sim simulators. We have submitted documentation PR on running PTX sims to Accel-sim. The project also helped us do a deep analysis of GPU architectures, especially that of Fermi, GPGPU workloads and their different characteristics. The literature survey done by us for this project helped us understand the landscape of GPGPU-sim research and learn about some key issues and some elegant solutions to them. In particular, we were able to learn more about the different types of caches used in GPUs and the significance of each of them. The code for our implementation

is made available open-source on Github and we hope it would be key to further research in this area.

9 Conclusion and Future Scope

Traditional caches are ineffective in extracting significant performance out of streaming data in GPGPU applications. The large data structures and reuse distances associated with such applications pollute the typical small caches used in GPGPUs, with zero-reuse blocks. There are certain software techniques that can be used to improve the efficiency of caches but they require a good amount of programming effort. The adaptive cache bypassing technique implemented here is a hardware mechanism that attempts to predict the zero-reusable blocks using a simple PC-based prediction table and bypasses them without bringing them into the L1D cache. This will increase the hit rate, the performance, reduce the execution time and power consumption of the system.

Important learnings from this project are:

- Helped us work hands-on on a simulator.
- Gave us a better understanding of the GPU architecture taught in class by examining the functions of each modules and how they interacted with each other.
- Taught us the importance of tailoring a structure towards the application that it is meant for, to extract the best performance boost out of it. Traditional Caches needs to be tailored according to the streaming nature of GPGPU workloads to get any performance benefit.
- The literature survey helped us understand how multiple people approached the same problem around the same time with different solutions.

The bypass predictor implemented was unable to extract a significant amount of performance speedup compared to the baseline, and we plan to look into the victim candidate selection part of the logic to debug this further. We plan to extend our study to analyze the effect of tuning parameters like threshold value, the prediction table initialization value, and different hash functions on the efficiency of the predictor. We further plan to investigate using the bypass predictor along with thread scheduling policies to reduce the congestion on on-chip resources like MSHR, fifo queue, etc. This will reduce the stalls in the system due to resource congestion by scheduling the threads in an intelligent way and improve the performance of the system further. We see merit in the bypassing technique discussed and would like to extend the design in the directions mentioned above, microbenchmark it and contribute towards the research community.

10 Individual contributions

1. Rajesh Shashi Kumar

   (a) Simulation bring-up on GPGPU-sim standalone and Accel-Sim.

   (b) Developed scripts for benchmarking and result collection using standalone GPGPU-sim.

   (c) Code and control understanding of GPGPU-sim.

   (d) Figured out a reliable way to obtain Program Counter value of cache access transactions.

   (e) Implemented the addition of metadata information in L1 and L2 caches by investigating interfaces in GPGPU-sim.

   (f) Implemented the threshold computation logic for bypass predictor and decrement counter logic for hits.

   (g) Added the L1-L2 communication for verification using storage and retrieval of bypass bit.

   (h) Generation and analysis of performance and power numbers of the baseline design (Fermi architecture) and the new design (Volta architecture).

   (i) Debugging the integration issues using gdb flow.

   (j) Creation of schematics and diagrams used in presentation and final report.

2. Aishwarya Lekshmi Chithra

   (a) Related Work Survey

   (b) Identified the files pertaining to the cache logic to be modified

   (c) Figured out the data flow of L1 access from *shader.cc* to *gpu-cache.cc*

   (d) Created the bypass prediction table and the logic to increment the counters

   (e) Added the logic to identify if there was a victim on MISS

   (f) Created the verification check with L2 *bypassbit*

   (g) Debugging the integration issues using gdb flow and fprintf statements.

   (h) Distilling key insights of this work in the lightening talk, presentation and report.

References

[1] Tor Aamodt. 2012. GPGPUsim. http://gpgpu-sim.org/

[2] Accel-Sim. [n. d.]. gpu-app-collection. https://github.com/accel-sim/gpu-app-collection/

[3] Xuhao Chen, Li-Wen Chang, Christopher I. Rodrigues, Jie Lv, Zhiying Wang, and Wen-Mei Hwu. 2014. Adaptive Cache Management for Energy-Efficient GPU Computing. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. 343–355. https://doi.org/10.1109/MICRO.2014.11

[4] Xuhao Chen, Li-Wen Chang, Christopher I. Rodrigues, Jie Lv, Zhiying Wang, and Wen-Mei Hwu. 2014. Adaptive Cache Management for Energy-Efficient GPU Computing. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. 343–355. https://doi.org/10.1109/MICRO.2014.11

[5] Wenhao Jia, Kelly A. Shaw, and Margaret Martonosi. 2014. MRPB: Memory request prioritization for massively parallel processors. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 272–283. https://doi.org/10.1109/HPCA.2014.6835938

[6] M. Kandemir, J. Ramanujam, M.J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh. 2004. A compiler-based approach for dynamically managing scratch-pad memories in embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 23, 2 (2004), 243–260. https://doi.org/10.1109/TCAD.2003.822123

[7] Mahmoud Khairy, Zhesheng Shen, Tor M. Aamodt, and Timothy G. Rogers. 2020. Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 473–486. https://doi.org/10.1109/ISCA45697.2020.00047

[8] Samira Manabi Khan, Yingying Tian, and Daniel A. Jiménez. 2010. Sampling Dead Block Prediction for Last-Level Caches. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. 175–186. https://doi.org/10.1109/MICRO.2010.24

[9] Dong Li, Minsoo Rhu, Daniel R. Johnson, Mike O'Connor, Mattan Erez, Doug Burger, Donald S. Fussell, and Stephen W. Redder. 2015. Priority-based cache allocation in throughput processors. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 89–100. https://doi.org/10.1109/HPCA.2015.7056024

[10] Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. 2012. Cache-Conscious Wavefront Scheduling. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. 72–83. https://doi.org/10.1109/MICRO.2012.16

[11] Yingying Tian, Sooraj Puthoor, Joseph L. Greathouse, Bradford M. Beckmann, and Daniel A. Jiménez. 2015. Adaptive GPU Cache Bypassing. In *Proceedings of the 8th Workshop on General Purpose Processing Using GPUs (GPGPU-8)*. Association for Computing Machinery, New York, NY, USA, 25–35. https://doi.org/10.1145/2716282.2716283

[12] Zhong Zheng, Zhiying Wang, and Mikko Lipasti. 2015. Adaptive Cache and Concurrency Allocation on GPGPUs. *IEEE Computer Architecture Letters* 14, 2 (2015), 90–93. https://doi.org/10.1109/LCA.2014.2359882