Simulation of prefetching mechanism for GPGPUs and understanding gpgpusim

Aarohi Patel, Munawira Abdul Kotyad, Ramapriya Namo Achyutha and Huiyang Zhou
Department of Electrical and Computer Engineering,
North Carolina State University
[ahpatel4, mkotyad, ramapr, hzhou] @ ncsu.edu

***Abstract* — this paper discusses about the prefetching mechanisms that can be applied to gpgpus. The prefetching mechanism used in this paper is already implemented and published in MICRO – 43 by Jaekyu et al.[1] and our implementation is based on this.**

Prefetching mechanisms have been proved to be an efficient way to hide memory latency in conventional CPUs. It helps to effectively use all the pipeline stages of a single processor system by prefetching data that would have otherwise lead to stalls in pipeline due to cache misses. However, given that there will be hundreds of threads that will be operating in parallel, straight forward mapping of prefetching will not be always beneficial and may harm performance. [1] Discusses techniques on how to judiciously prefetch while not hampering genuine demand memory requests. We focus on implementing hardware prefetcher training algorithm as described by [1] on tag based cache model [2]. We also describe our understanding of gpgpusim [3] infrastructure in detail.

Keywords—hardware prefetcher, gpgpusim

I. INTRODUCTION

The paper "Many-Thread Aware Prefetching Mechanisms for GPGPU Applications" [1] describes the concept of Many-Thread (MT) aware prefetching as both software and hardware mechanisms. Many-Thread Aware Software Prefetching consists of 2 components – stride prefetching and Inter thread prefetching (IP). Stride prefetching is similar to traditional prefetching mechanism with a prefetched cache storing all prefetch requests. IP is a new concept introduced by the above paper and discusses about co operative prefetching amongst threads. i.e., one thread from a particular warp prefetches data for thread with same index in another warp. Many – Thread aware hardware prefetching – MT-HWP – extends the stride prefetcher implementation by introducing additional prefetch structures like Promotion table/ Global stride table, IP table and per warp stride table (PWS). By training these tables based on stride behavior of incoming instruction stream, accurate prefetching can be done.

Even with these enhanced training algorithms for prefetching, prefetched data is not always useful and hurts performance in some cases even with 100% accurate prefetches. In order to overcome these negative effects, [1] describes feedback mechanism – Adaptive Prefetch Throttling by which decision to increase/ decrease or stop prefetching is made.

Our work focuses on simulating the MT-HWP of [1] using a tag based cache architecture. We try to run application benchmarks like 2D convolution, matrix multiplication and so on extracted from the gpgpusim. We also present our understanding about the simulator at the software level by explaining about the classes that represent various architectural structures of the GPGPU front end and data path.

This paper is organized as follows: Section II describes in detail the understanding of the gpgpusim infrastructure. Here we try to map various aspects of GPGPU architecture with corresponding classes in C++. It also describes about the configuration file which can be used to fine-tune the parameters for the simulator. Section III describes code of our simulation of MT-HWP. Section IV explains the methodology about our implementation and discusses the experiments conducted. We also mention about how we generated benchmarks to test this functionality. Section V concludes our work and Section VI points to future work that can be carried out.

II. UNDERSTANDING GPGPUSIM

The paper [1] implements MT prefetching techniques using a cycle accurate simulator called macsim [4] for memory intensive benchmarks. We try to understand how the implementation can be mapped onto future versions of gpgpusim. We start by first understanding the gpgpusim model.

GPGPUsim consists of 2 models – Timing model and Power model. The timing model gpgpu-sim, models the cycle accurate timing simulator. The correctness of the results from gpgpu-sim is ensured with the use of functional simulator that validates outputs for input traces. GPUWattch gives the power model. We will split the section into 3 parts as shown in figure below. First part explains the shader core implementation that corresponds to the streaming processors (SPs) in the GPU architecture. Second part describes about the pipelined SIMD data path consisting of ALU and memory operations with particular stress on the GPU caches. Third part explains the files related to configuring the simulator for various architectural parameters.

GPU cores:

The file shader.cc/h has classes and functions that architecturally maps to SPs, Scoreboard, Operand collector unit, SIMD execution units (modeling the execute stage), Load

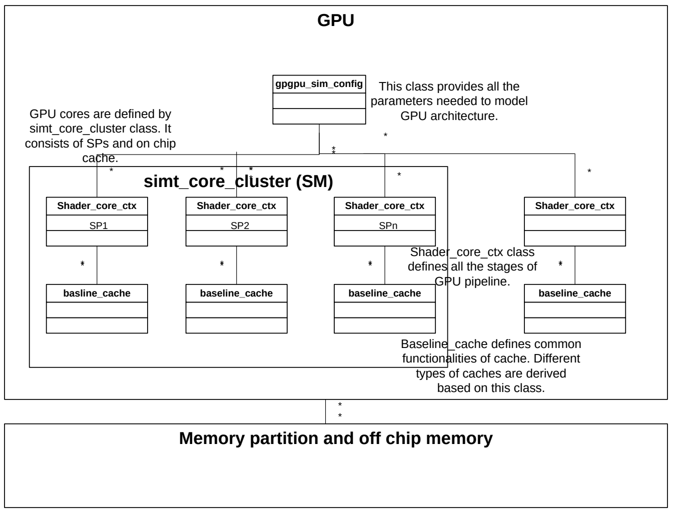

GPU
gpgpu_sim_config
This class provides all the parameters needed to model GPU architecture.
GPU cores are defined by simt_core_cluster class. It consists of SPs and on chip cache.
simt_core_cluster (SM)
Shader_core_ctx
SP1
Shader_core_ctx
SP2
Shader_core_ctx
SPn
Shader_core_ctx
Shader_core_ctx class defines all the stages of GPU pipeline.
basline_cache
baseline_cache
baseline_cache
baseline_cache
Baseline_cache defines common functionalities of cache. Different types of caches are derived based on this class.
Memory partition and off chip memory

Fig 1: Class: Architecture Overview

and Store units and Write back stage. Some of the important points regarding nomenclature of variables are as follows:

- If the variables are related to threads, they are named as n_*
- If the variables are related to thread blocks, warps etc., they are named as m_*

Class thread_ctx_t

This class contains information for maintaining status per thread such as

- cta_id – Indicates the SM to which the thread belongs.
- Other statistics such as total number of instructions per thread, number of L1 accesses, read and write misses per thread etc.,

Class shd_warp_t

This class contains per warp information along with functions that are described below:

1. The constructor contains the warp size and which SP (i.e., shader) the warp belongs to. It also initializes the number of outstanding stores and instructions in pipeline to 0.
2. *reset( )* – Resets certain parameters related to warp such as warp completion, last fetch etc.,
3. *init( )* – Initializes cta_id (thread id), warp_id, next_pc, number of active threads.
4.
   a. *Set completed( ), get_n_completed( )* – getter and setter function for the number of threads completed.
   b. *get_n_atomic( ), inc_n_atomic( ), dec_n_atomic( )* – returns, increments/decrements the number of instructions.
   c. *get_pc( ), set_pc( )* – getter and setter functions for the next pc.
5. *ibuffer_fill( ), buffer_empty( ), ibuffer_free( ), ibuffer_flush( ) , ibuffer_next_valid( )* – These are related to the Instruction buffer and fills, flushes or checks whether the ibuffer is empty or not.
6. *inc_store_req( ), dec_store_req( ), stores_done( )* – These functions increment, decrement the number of stores or mark all the stores to be completed.
7. *set_membar( ), clear_membar( )* – sets or clears the memory barrier, if set then warp is waiting at a memory barrier.

Class scheduler_unit

This class defines the scheduler unit. The constructor contains information related to scheduling the instructions. It consists of registers for SP, SFU and memory units, object of Scoreboard class, functions to supervise warps, object of shader core and object of simt stack.

Class LooseRoundRobinScheduler

This class is derived from the scheduler unit and used when the scheduling policy is round robin.

Class TwolevelScheduler

This class is derived from Scheduler unit and is used when the scheduling policy is Two-level Scheduling.

Class Operand Collector

This class is a operand collector based register file unit and contains functions to modify the number of input and output ports issuing in SP, SFU and LD/ST unit. It contains various subclasses that are as mentioned below:

1. *Class Opt_t* – The class contain functions like
   a. *get_wid( )* – gives warp id.
   b. *get_active_count()* – gives the count of active warps and collector units.
   c. *get_active_masks()* – gives active masks for operators and collector unit.
   d. *get_bank()* – gives the respective bank number for accessing memory operation.
   e. *get_operand()* – gives the operand for the instruction.
2. *Class arbiter_t*
   This class contains variables related to accessing the register file banks and allocates read and write access, information on number of banks, collectors etc., It contains functions to know whether the bank is idle. *allocate_bank_for_write(), allocate_bank_for_read()* functions allocate banks for read/write operations.
3. *Class Allocation_t*
   Links a read or write request generated by op_t to an allocated access by using the class arbiter_t.
4. *Class input_port_t*
   The class defines the inputs which are given to the operand collector.
5. *Class Collector_unit_t*
   This class models the collector unit per warp and collects the operands. Functions - *collect_operand(), get_num_operands(), get_num_regs()* help to do so .
6. *Class Dispatch_unit_t*
   If all the operands in the respective collector units for each warp are ready, it dispatches the instructions.

Class barrier_set_t

This class contains functions to allocate; set and de allocate the barrier. It also suggests which warp is free for fetch or when the individual warp does hits the barrier.

Class simd_functional_unit

This class models the SIMD functional units.

Class pipelined_simd_unit

This class is derived from simd_functional_unit. This class checks whether the pipeline is free or not and returns information about the number of active lanes in the pipeline.

Classes sfu, spu and ldst_unit

These are derived from pipelined_simd_unit and define the ALU and memory operations.

Class shader_core_config

This class contains configurations for a Shader core (SP). It records data such as number of threads, number of warps, clock gating, number of warps per scheduler, max number of thread block per SP. It also ensures that number of threads in shader is not more than that configured in the hardware abstract model. It configures shader core resources like number of register banks, local memory map, registers used per warp, sp and sfu latency.

Class shader_core_stats_pod

This class defines various variables for getting the power related statistics.

Class shader_core_ctx

This class models SIMT core and the pipeline stages

1. *cycle( )* – This is a very important function and models various pipeline stages. It is a cycle based implementation of fetch, decode, register read, schedule, execute and write back stages.
2. *issue_block2core( )* – brings in blocks of data from the kernel.

Class SIMT core cluster

This class models the SIMT core cluster i.e., SM of the GPU architecture.

Class shader_memory_interface and perfect_memory interface

These classes model the memory interfaces and are derived from the class mem_fetch_interface.

Memory Model:

The files gpu-cache.cc and gpu-cache.h define the on chip memory model for the GPU architecture. It consists of hierarchy of cache model starting with baseline cache which supports common cache functionalities. Different types of caches like read only or constant memory, texture, L1/L2 and shared memory are derived from the baseline cache class. Detailed description of these memory classes are as follows:

Enumerated types:

These types indicate various cache policies and states and helps in easy understanding of the architectural state the cache might be in. gpu-cache.h defines the following enumerated types - cache_block_state, cache_request_status, cache_event, replacement_policy_t, allocation_policy_t, write_policy_t, write_allocate_policy_t and mshr_config_t.

Structure cache_block_t

This structure defines and allocates time stamp details like last cache fill time, last access time and address details like tag of the cache block.

Class cache_config

This class models a cache for the given set of architectural parameters. It specifies replacement, write, allocation policies and miss status handling register type. It contains a set of getter setter functions to access number of cache lines, size of cache line, index, tag and block address. The configuration details set in this class are used in defining various cache types like constant cache, texture cache, L1/L2 data cache etc.,

Class tag_array

This class implements the cache management operations like accessing a cache, deciding on hit/miss, modifying coherence states, identifying line to replace and so on.The member functions are explained below:

tag_array::probe(new_addr_type, unsigned) – For a given address, this function checks to see if the tag is present in cache (if it is hit/ pending). If it is a hit/pending, then it returns the index and status of that cache line. The function also keeps track as to which line to replace next based on either LRU / FIFO policy. If there is a cache miss and there is no space to allocate the new line, then a RESERVATION_FAIL state is returned.

tag_array::access() – This function keeps record of all types of accesses to the shader core cache like total number of accesses, number of hits, number of misses and number of pending hits.

tag_array::fill() – This function is responsible for allocating a cache line for a particular address along with the timestamp information.

tag_array::flush() – This function invalidates a cache block to maintain coherency.

tag_array::windowed_miss_rate() – This helps in tracking the miss rates for a particular length of an instruction stream.

tag_array::new_window() – This starts to keep track of new window of instructions by storing previous window's number of misses/accesses. These values are needed to calculate the next windowed miss rate.

Other tag-array member functions like print_stats() and get_stats() are required for printing out the final total number of accesses/misses/hits details.

Class mshr_table

This class keeps track of all the in flight memory accesses that will help in servicing independent memory operations in out of order fashion. The member functions are explained as below:

mshr_table::probe() – For a given block address, this function checks to see if there are pending requests for the same address to lower memory levels.

mshr_table::full() – This function checks if new memory accesses can be accommodated for tracking by returning table full status.

mshr_table::add() – This function will add new memory access to the table or merge to an already existing access for the same block address. Critical operations like atomic operation can also be indicated along with the entry using this function.

mshr_table::mark_ready() – This function is defined for the response side of a memory operation. It checks for the block address and accepts the new cache fill response. This way an entry can be marked to be ready for processing. At this time, address related to atomic operations is also indicated.

mshr_table::next_access() – All the cache responses are first updated to be ready and stored in a queue. This function returns the next ready access to be processed. It then releases the entry from the MSHR table.

Class baseline_cache

This class forms the base class that implements all the common functionalities associated with the cache. It is at the top of cache model hierarchy followed by all other cache types like read-only, data cache and texture cache.

baseline_cache::cycle() – This function sends the next request to lower level of memory by popping from front of the queue and pushing the request onto free memory port.

baseline_cache::fill() – This function forms the interface for response from lower memory level. It calls the tag_array fill() function to update the incoming cache line along with function to mark ready for processing in MSHR table.

baseline_cache::waiting_for_fill() – This function checks to see if a memory fetch is waiting to be filled by an incoming cache line from lower memory.

baseline_cache::send_read_request() – Method overloading is used to define 2 variants of operations. One defines read miss handler without writeback and other by checking if MSHR hit of if MSHR is available.

Class data_cache

This class is derived from the baseline_cache and implements various read/write policies that can be set using the config file. The below table defines these functions.

Classes l1_cache and l2_cache

These 2 classes are derived from the data_cache class and implements access functions to send requests to lower memory levels. These are meant to model multi level cache hierarchy for Fermi architecture. l1_cache models global write evict and local write back policy while l2_cache models shared cache with global write-back and write-allocate policies.

GPU configuration:

The file gpu-sim.cc/h is used to set all the architectural parameters for the entire GPU. It sets the number of core clusters, configuring clock domains, configuration related to merging scatter-gather requests and so on.

reg_options( )

This method contains the object of class option parser and all arguments to the simulator are passed through this class.

Struct_memory config

This structure carries properties related to memory configurations.

Class gpgpu_sim_config

This class is derived from classes power_config and gpgpu_functional_sim_config and contains various configuration parameters.

The structures *shader_core_config, memory_config ,get SIMT_core_cluster* return the configurations of the shader core used by functional simulator. There are functions for checking the deadlock, various performance counters, etc., Function *issue_block2core( )* launches a CTA (thread block). Function *dump_pipeline( )* shows the payload at every stage of the pipeline and can be useful in debugging.

Write-Hit	
wr_hit_wb	This method marks the block as modified and updates LRU state
wr_hit_wt	Sends the write request directly to lower memory level
wr_hit_we	Sends write request to lower memory level and also invalidates the corresponding block
wr_hit_global_we_local_wb	In case of private caches, this method implements write back for local memory while write evict for global memory

Write-Miss	
wr_miss_wa	In case of a miss, this method sends write request to lower memory level and also allocates the block by sending a read request for the same block
wr_miss_no_wa	This configuration does not allocate a cache block on write miss but simply sends the write request to lower memory level

Read-Hit	
rd_hit_base	This config updates the LRU status in case of hit. However in case of special atomic instructions the block is marked as MODIFIED. Atomic operations here are treated as global READ/WRITE request.

Read-Miss	
rd_miss_base	In case of a miss, this config sends read request to lower memory level. If the block to be evicted is modified then write-back is performed as specified.

Fig: 2,3,4 – Various Read/Write Configurations

III. CODE IMPLEMENTATION

We have implemented a C++ application that emulates the prefetcher algorithm explained in the paper [1]. The prefetcher algorithm uses 3 different stride training tables – Inter-thread Prefetcher Table(IP), Global Stride Table (GS) and Per Warp Stride table (PWS). The decision logic helps to select a stride from one of the tables.

All the tables used are PC based stride prefetchers. They calculate strides for memory accesses for the same instruction but different threads across all warps. In data intensive applications, the memory addresses accessed are generally in an increasing pattern for the same PC. The PC based stride prefetcher takes advantage of this access pattern to prefetch the data.

The algorithm used by the three tables is described in detail here.

1) Inter-thread Prefetcher table:

The Inter-thread Prefetcher table is used for calculation of strides across warps and across threads. There is one IP table which trains strides for all threads across multiple warps. The table is accessed using the pc.

- If the pc is not found then a new entry is made and the last thread and last address fields are updated.
- If the pc is found and the pc is trained, then the trained stride is returned which is used to prefetch the next addresses.

Current stride is calculated using this formula

(current mem_addr – last_addr) / (current thread id – last thread id)

This stride is now updated in the stride counter table. Only when a particular stride has a stride count greater than 2, will the pc be considered trained. The stride counter keeps the 3 most recently used strides and their counts in the stride table. The stride that has the max count is updated as the trained stride as shown below:

PC	Trained	Trained stride	Last address	Last thread ID	Stride counter	
PC1	True	100	5000	2	Stride	Stride count
					100	3
					200	2
PC2	False	-	2000	3	Stride	Stride count
					100	1

Fig 5: IP Table

2) Per Warp stride table

The Per Warp Stride table is used for calculation of strides across threads but within a warp. The algorithm for this table is exactly the same as that of IP table. Each warp has its own per warp stride table.

3) Global Stride table

PC	Stride

Fig 6: GS Table

The Global stride table is used to calculate strides across warps. An entry is made in the Global Stride table only after promotion from the PWS tables. Every time any PWS table is accessed, the GS_promote() function is called to check for a qualifying GS entry.

The GS_promote() call iterates through all the PWS tables and checks if the current PC has the same trained stride in atleast three of the PWS tables. If there are more than one stride with a count greater than two, then the stride with the maximum repetition count is chosen. This is updated in the GS table.

The GS table reduces the amount of time required for stride access. If an entry is found in the GS table, the stride calculation ends and the next prefetch request is made with the current stride. After a GS entry is made then even if a particular PWS for a warp does not have an entry for a given PC, it can directly access the GS table for the PC and create a prefetch request.

All three tables use the LRU replacement policy.

Design Logic:

Whenever a memory access is made, the prefetcher logic is used to calculate the prefetch stride. The prefetcher logic works as follows:

- The GS and IP tables are accessed. If there is a hit in both the tables then the GS table is given higher priority because the entries in the GS table have been trained for a longer time and hence the confidence level is higher.

- If both tables miss then the PWS table is accessed. If there is a hit then the stride is returned and the stride table is updated. The GS_promote() logic is used to promote a stride entry among warps if it qualifies.

The stride is used to calculate the prefetch address and multiple blocks are prefetched based on the prefetch width.

IV. EXPERIMENTS AND RESULTS

Implementation of MT-HWP is done in C++ on top of a tag based cache simulator. Our simulation takes PC address, memory address being accessed, warp ID and thread ID as inputs. Baseline model used is just the cache simulator with no prefetching.

Benchmarks: To run experiments, We required memory access patterns shown in real world data intensive applications. To create such benchmarks, we extracted memory access information from the following benchmarks: matrixMultiplication, 2DConvolution, BlackScholes and Histogram.

Performance results with the above benchmarks are as explained below:

Useful Prefetching:

1. Histogram

For this trace, the prefetcher algorithm showed a slight improvement in the number of misses. The number of misses reduced with increasing prefetch width and at a particular value of prefetch width, the performance remains a constant.

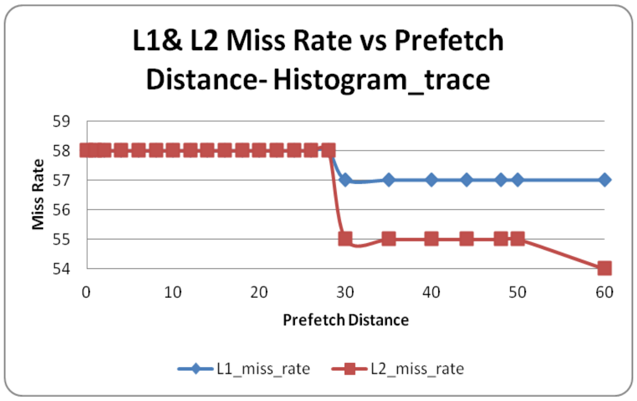


Fig: 7

2. Self generated

Since the above benchmarks did not show the memory access pattern that was required to show the performance of our prefetcher simulator, we created a trace that shows a strided memory access pattern. This trace resembles the sort of memory access pattern that a data intensive application would have. When we ran the prefetcher on this trace we found a huge difference in the number of misses. The number of misses reduces as we increase the prefetch width and then hits a standstill. Thus, beyond a point increasing the prefetch width does not make any difference. This was the expected performance from the prefetching algorithm.

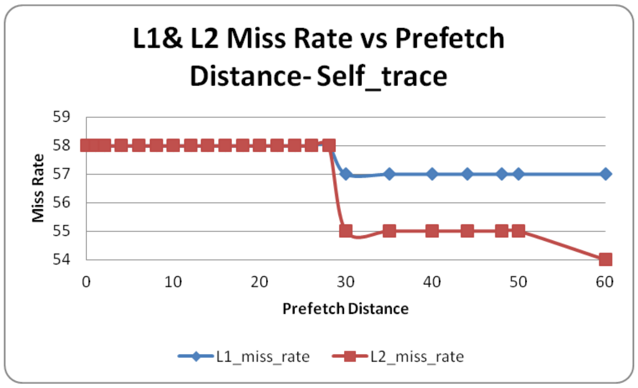
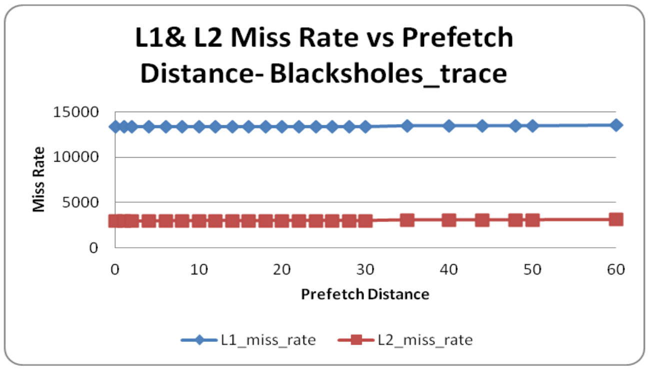


Fig: 8

Harmful Prefetching:

1. Blacksholes, 2DConvolution and Matrix Multiplication

For these traces the performance degraded after implementing prefetching, this might be due to eviction of useful data by useless prefetches. Thus, we could not show the expected output from these traces.



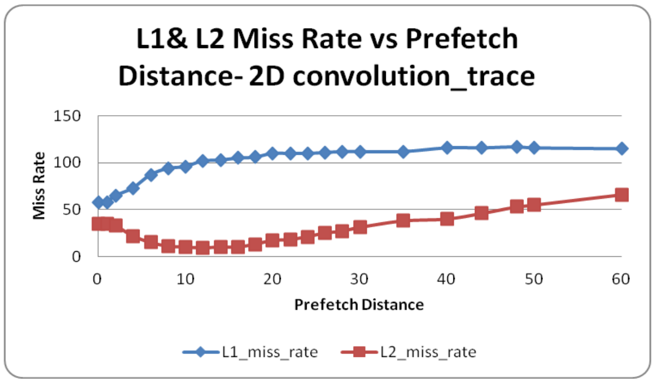


Fig: 9,10

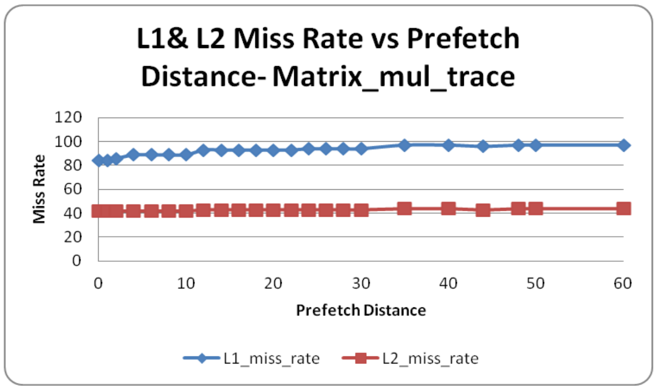


Fig: 11

V. CONCLUSIONS

We observed a significant difference in the number of misses for applications that showed strided memory accesses. These are the kind of applications that we were targeting with our stride based prefetcher implementation. For applications that did not show a strided memory access pattern, the performance degraded on using prefetching. Thus, if used with the correct applications, our prefetching implementation will give a considerable improvement in performance and help reduce the memory latency of such applications.

VI. FUTURE WORK

Our work only simulates the functional aspects of MT-HWP on standalone cache architecture. Also, the benchmarks that were used were for common applications. As future work, implementation can be extended to be cycle accurate and integrated on the gpgpusim. This work would enable the study of more real world effects of prefetching for GPGPUs.

ACKNOWLEDGMENTS

We gratefully acknowledge the support of Dr. Huiyang Zhou, our professor from Department of Electrical and Computer Engineering at North Carolina State University for providing this learning opportunity as part of the *Advanced Computer Architecture: Data Parallel Processors* coursework and Ping Xiang, TA for the course, for helping to generate benchmarks for this project.

REFERENCES

[1] Jaekyu Lee; Lakshminarayana, N.B.; Hyesoon Kim; Vuduc, R., "Many-Thread Aware Prefetching Mechanisms for GPGPU Applications," *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on* , vol., no., pp.213,224, 4-8 Dec. 2010
doi: 10.1109/MICRO.2010.44

[2] ECE521 and ECE786 coursework offered by Department of Electrical and Computer Engineering, North Carolina State University

[3] Ali Bakhoda, George Yuan, Wilson W. L. Fung, Henry Wong, Tor M. Aamodt, Analyzing CUDA Workloads Using a Detailed GPU Simulator, in IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Boston, MA, April 19-21, 2009.

[4] https://code.google.com/p/macsim/