GPU RESEARCH PROJECT DOCUMENTATION

Two-level Warp Scheduling Utilizing Dynamic Warps

By:

AMIRALI ABDOLRASHIDI

DEVASHREE TRIPATHY

SHAHRIYAR VALIELAHI ROSHAN

Fall 2015

Abstract

Warp scheduling is a vital stage in a GPU's execution cycle. Since the program counter for all warps differ, the scheduler needs to take into account the number of available resources and ready operands before issuing the instruction to one of the execution units. In recent years, with the rise of GPGPU's popularity, numerous attempts have been made to improve the scheduling process, such as dynamic warps, more complex scheduling queues and hiding memory access latencies.

Table of Contents

I. Introduction

Graphics processing units (GPUs) have been present in the computing industry for over 40 years. Their special structure allows us to perform many operations simultaneously. However, the resources on the GPU are limited and some operations take much more time than the others. To manage the execution of the warps, a scheduler should keep a list of warps that can be issued at any time and issue them in an order that maximizes the GPU's resource utilization and performance.

Our project is based on a paper by Narasiman [1], which proposes to improve the warp scheduling process using a two-level round robin scheduler and a large-warp architecture. We have also utilized a two-level round robin scheduling process, albeit coupled with checking all warps in a fetch group for being busy with memory instructions before switching the current fetch group. This project also utilizes dynamic warps to reduce branch divergence. The simulation of this work has been done on GPGPU-Sim [2] with several benchmarks such as BFS and Hotspot.

II. Related Work

Our primary focus in this project is [1], where Narasiman proposes a two-level round-robin warp scheduler and a large-warp mechanism to improve the latency and reduce warp divergence respectively. When combined, they outperform the regular round robin scheduler by over 19%. An improvement was made in **Error! Reference source not found.**, where Jog proposes the addition of a CTA-aware scheme in the warp scheduling, increasing the warp locality and also the cache and memory hit rates, improving the speed by another 19%.

Many improvements in GPU performance focus on the memory and cache access. Among recent works, in [8], Mao proposes a versatile warp scheduling which also improves the cache hit rate via increasing the cache locality and that of the cooperative thread array (CTA), increasing the performance by 80.7% compared to an LRR scheduler. Also in **Error! Reference source not found.**, Li has proposed an efficient locality-monitoring mechanism to dynamically filter the data copied into the cache in such a way that the data with high reuse and short reuse distances are stored in the L1 D-cache, achieving a 30.3% performance improvement over the baseline architecture.

In [3], Yu explores the thread level parallelism (TLP) at the run-time using a trigger-based method. It is noteworthy that the role of the pipeline efficiency in the TLP optimization is also highlighted. The resulting technique, called SAWS (Stall Aware Warp Scheduler), has achieved an improvement of 115.2% speedup compared to loose round robin (LRR) scheduling.

In [3], Lee addresses the "warp criticality" issue and proposes several algorithms to balance the warps' execution frequencies based on a criticality prediction scheme, thereby increasing the performance by 17% on average compared to regular round robin scheduler.

In [7], Rogers proposes variable warp sizing which aims to decrease divergence, improving the performance by 35% compared to the conventional 32-threaded-warp GPUs.

III. Implementation

We have used GPGPU-Sim to implement and simulate the mechanism of our warp scheduler. It is primarily comprised of two major parts; two-level round robin scheduling and dynamic warp creation, both of which are described below.

The entire design takes place in the file "shader.cc". Also there is a new function added in "shader.h" used for dynamic warps.

a. Two-level round robin warp scheduling

The LRR scheduler assigns equal priority to every slot reserved for a warp in a scheduler and issues them in order. For instance, the instruction in slot 0 will be issued first, followed by another in slot 1, after which the one in slot 2 is issued and so on. When the pointer reaches the final slot, it will be reset back to 0.

A two-level round robin scheduler is very similar in concept to the regular one. It includes several fetch groups with different instructions for different warps. The issuing pointer remains in the same group, issuing the corresponding instructions in a round robin fashion, while the instructions in other groups wait. This continues until all the warps in the scheduler are stalled due to high-latency memory access instructions. Only then will it proceed to switch to the next fetch group. The switching between the fetch groups is also round robin, hence the name.

In our implementation, we changed the function "order_lrr()" to override the regular round robin scheduling scheme, effectively replacing it with our own two-level round robin. In

every function launch, we determine the next index based on the two-level round robin and the current fetch group and populate the priority queue for the issuing accordingly,

We included a global switching variable called “queueOffset” that is responsible for fetch group switching which operates in the function “scheduler_unit::cycle()”. We check the warps in our current fetch group in this function and increase “queueOffset” which will later switch the fetch group when “order_lrr()” is launched again. Below you may see the code for the two-level round robin scheduler in “shader.cc”.

```
/*GPU-217:2 LEVEL ROUND ROBIN BY AMIRALI*/
//Begin the custom part
const unsigned int numOfGroups=NUM_OF_FETCH_GROUPS;
const unsigned int fetchGroupSize=(input_list.size()-1)/numOfGroups+1;//GPU-217 fetchGroupCeiling

        iterTemp = ( queueOffset==0 && (last_issued_from_input ==  input_list.end() || (
        last_issued_from_input-input_list.begin()+1)%fetchGroupSize==0) ) ? last_issued_from_input
        - fetchGroupSize + 1 : last_issued_from_input + 1 + queueOffset;

queueOffset=0;

//End of custom part
```

```
/*GPU-217:----START--------- modified 2 level RR, Stalling of warps by DEVASHREE and AMIRALI*/
tempFetchCounter=0;
unsigned fetchGroupSize= (m_supervised_warps.size() + NUM_OF_FETCH_GROUPS- 1)/NUM_OF_FETCH_GROUPS;
for(unsigned i= 0; i< MIN(fetchGroupSize,m_next_cycle_prioritized_warps.size()) ; i++) //
iterates through the warps in the same fetch group which is currently executing
]{
std::vector<shd_warp_t*>::const_iterator iter2= m_next_cycle_prioritized_warps.begin();
if( *(iter2 + i) !=NULL )
]{
unsigned warpId2= (*(iter2 + i))-> get_warp_id();

const warp_inst_t *pI1 = warp(warpId2).ibuffer_current_inst();
if((warp(warpId2).ibuffer_current_inst()) != NULL )
if(!issued_inst && (*(iter2+i))-> get_n_completed()!= MAX_WARP_SIZE && (pI1-> op == LOAD_OP ||pI1
-> op == STORE_OP ||pI1-> op == MEMORY_BARRIER_OP))
//iter2->op
{ tempFetchCounter++;}
}
}
```

b. Dynamic warp creation

Divergence is a very important factor in the performance of a GPU. Since all threads in a warp must execute the same instruction, when they are divided in a conditional branch, the warp takes at least twice the time to complete. This can lead to a dramatic decrease in total execution time if the divergence is not minimized.

Dynamic warps are created by populating the empty threads of a warp using threads from other warps in the queue. This results in more resource utilization for executing warps, thereby reducing divergence. In our implementation, we added a variable ("dyn_mask") for the current active mask in the "scheduler_unit::cycle()" function. First we copy the mask value of the current warp to "dyn_mask". Then we check all warps with the same instruction and transfer the corresponding missing thread from any other warp to the current warp we are about to issue, clearing the said mask bits in the source warp in the process. Then we issue the warp with the new mask.

Below is the code for the dynamic warp creation section of our code, also in "shader.cc".

```
/*GPU-217: LARGE DYNAMIC WARP CREATION by AMIRALI n DEVASHREE---------------START*/

//Find the warp with the same PC
unsigned pcLW= pc,pcLWTemp,rpcLWTemp;
for (std::vector< shd_warp_t* >::const_iterator iterTemp=iter+1; iterTemp <
m_next_cycle_prioritized_warps.end(); iterTemp++)
{
        if ( (*iterTemp) == NULL || (*iterTemp)->done_exit() ) {
            continue;
        }
    unsigned i=(*iterTemp)-> get_warp_id();
    if(!warp(i).waiting() && !warp(i).ibuffer_empty())
    {
        m_simt_stack[i]->get_pdom_stack_top_info(&pcLWTemp,&rpcLWTemp);
        if(pcLW==pcLWTemp)
        {
            source_mask=m_simt_stack[i]->get_active_mask();
            for(unsigned j=0;j<source_mask.size();j++)
                if(!dyn_mask[j] && source_mask[j])
                {
                    //Set the mask bit in the dynamic warp
                    dyn_mask.set(j,true);
                    //Reset the mask bit in the source warp
                    source_mask.reset(j);
                        }
        }
    }
```

IV. Results

We have run several benchmarks on our modified GPU model to test its performance. We copied the configuration files from the "GTX480" folder into the benchmark folder where the executables are. Benchmarks include Breadth-First Search (BFS), Hotspot (HS), Histogram

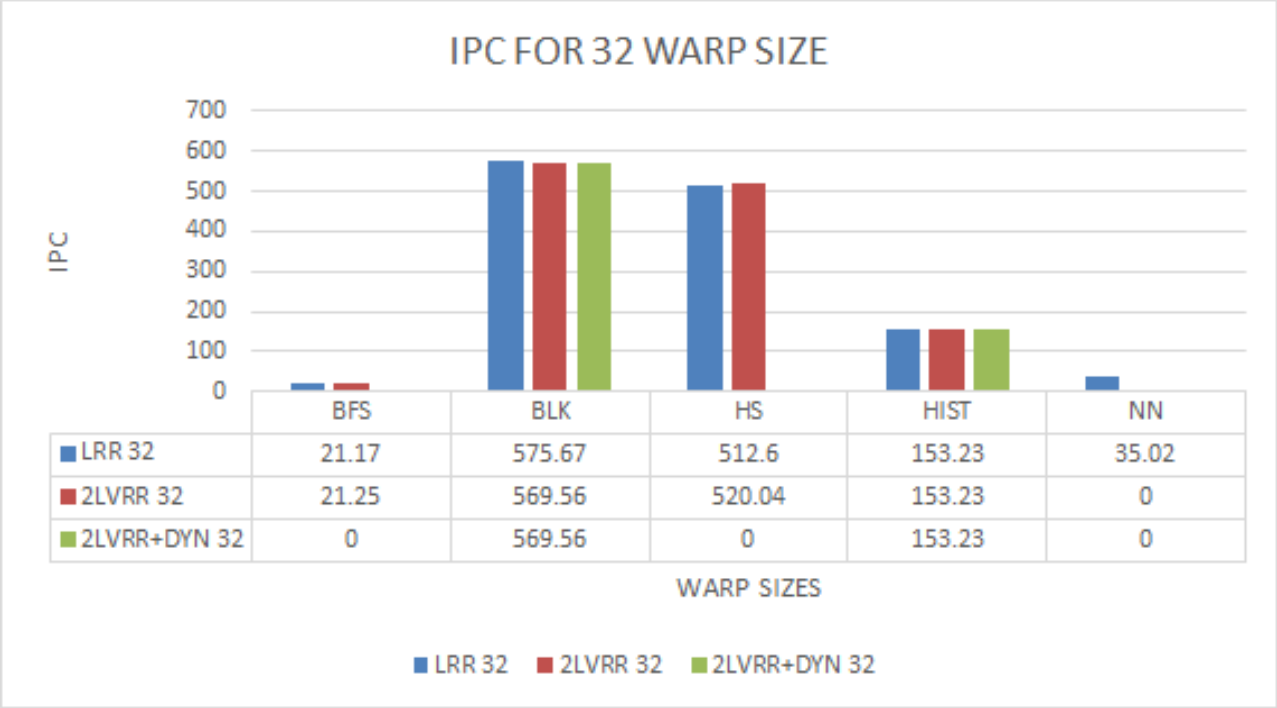
(HIST), Blackjack (BLK) and Neural Network (NN). Various sensitivity analyses were performed by simulating the LRR, 2-level RR and 2-level RR with dynamic warp scheduling for warp sizes of 8, 16 and 32. The simulation time and IPC for our five benchmarks are brought in the table below.

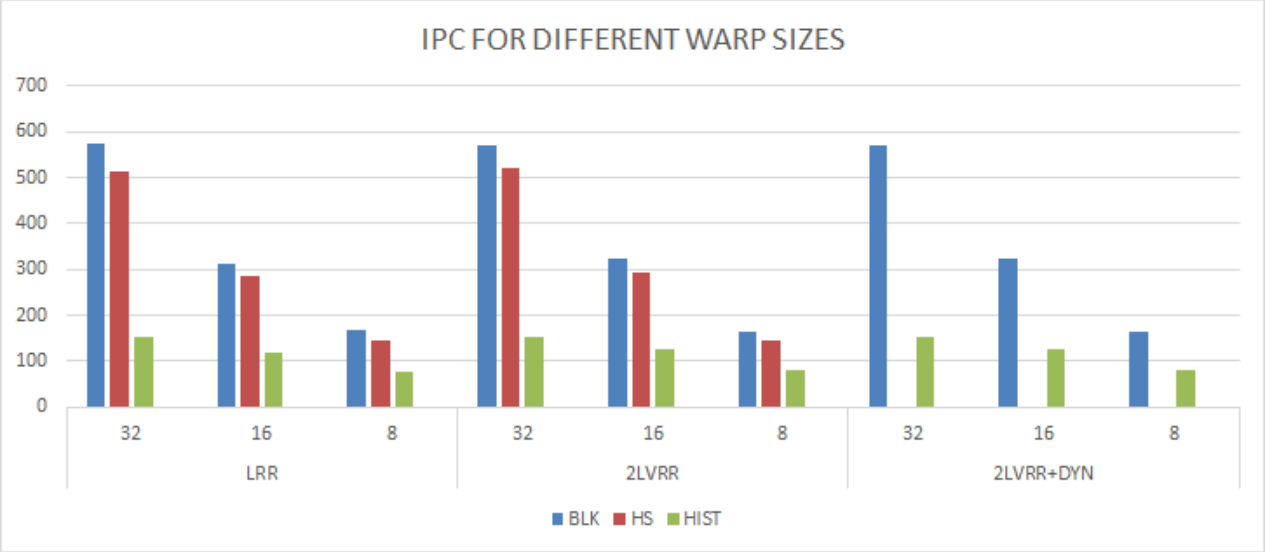
	Analysis	**LRR**			**2LVRR**			**2LVRR+DYN**		
	Warp Size	32	16	8	32	16	8	32	16	8
BFS	Time	10	12	16	10	13	17	-	-	-
	IPC	21.17	19.78	16.58	21.25	19.81	17.15	-	-	-
BLK	Time	402	653	1000	451	506	1027	376	774	1744
	IPC	575.67	314.16	166.72	569.56	324.40	162.97	569.56	324.39	162.97
HS	Time	287	350	601	268	297	666	-	-	-
	IPC	512.60	287.41	147.02	520.04	292.60	145.52	-	-	-
HIST	Time	2	2	3	2	2	4	2	3	6
	IPC	153.23	119.80	78.00	153.23	125.96	80.61	153.23	125.90	80.60
NN	Time	812	851	950	-	769	1003	-	-	-
	IPC	35.02	34.07	32.48	-	34.21	32.50	-	-	-

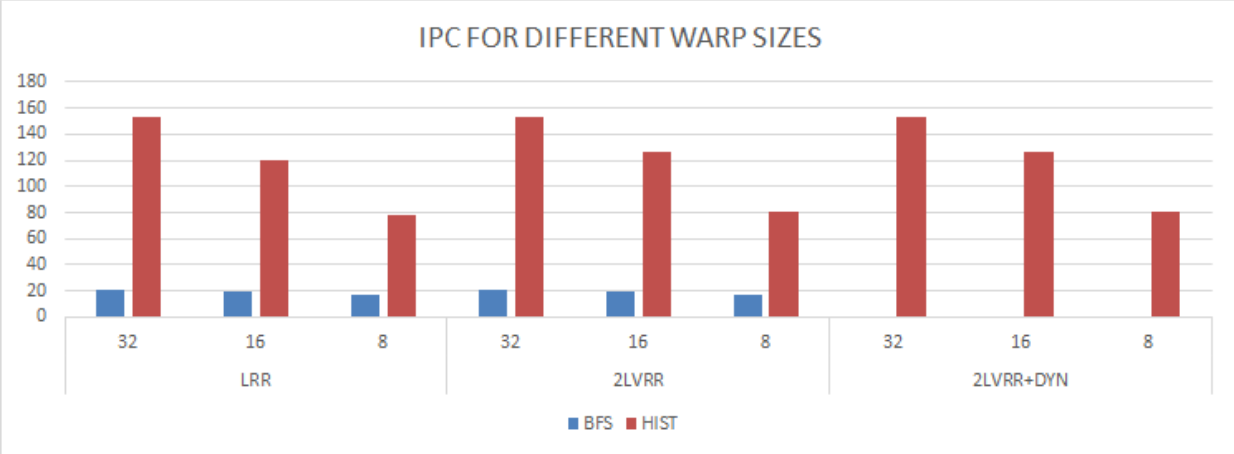
Note that in the case of dynamic warps, the benchmarks NN, BFS and HS fail due to the following error: “cuda-sim.cc: 1196: void ptx_thread_info::ptx_exec_inst (warp_inst_t&, unsigned int): Assertion inst.pc==inst.pc failed”. The NN also fails for two-level scheduling and warp size of 32 due to a segmentation fault. These are where the blank spaces in the table come from.

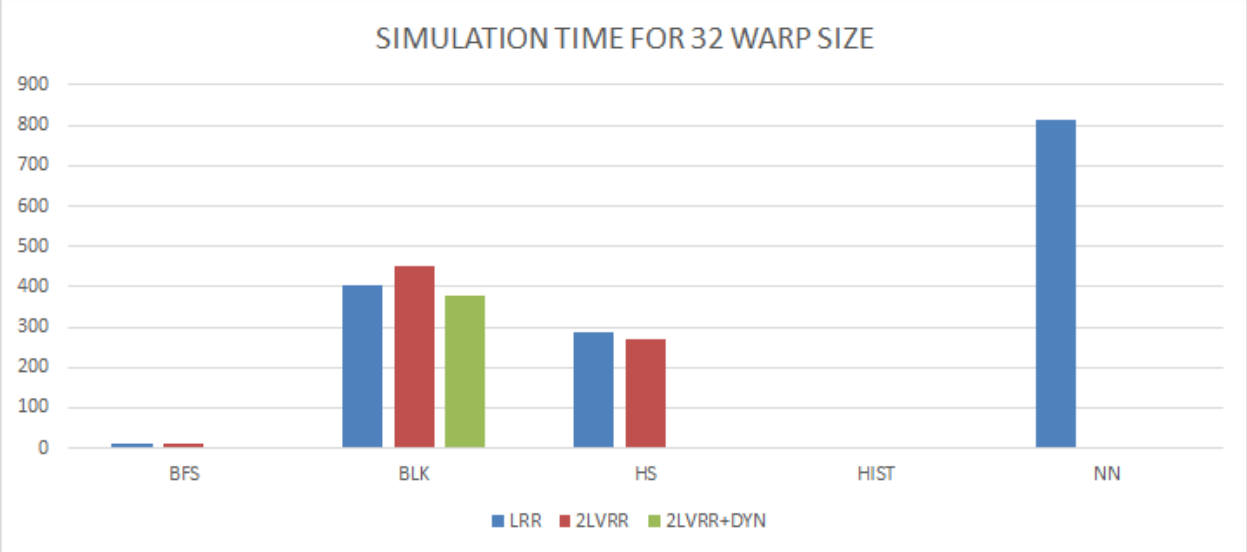
To change the warp size, three lines of the source files should be changed:

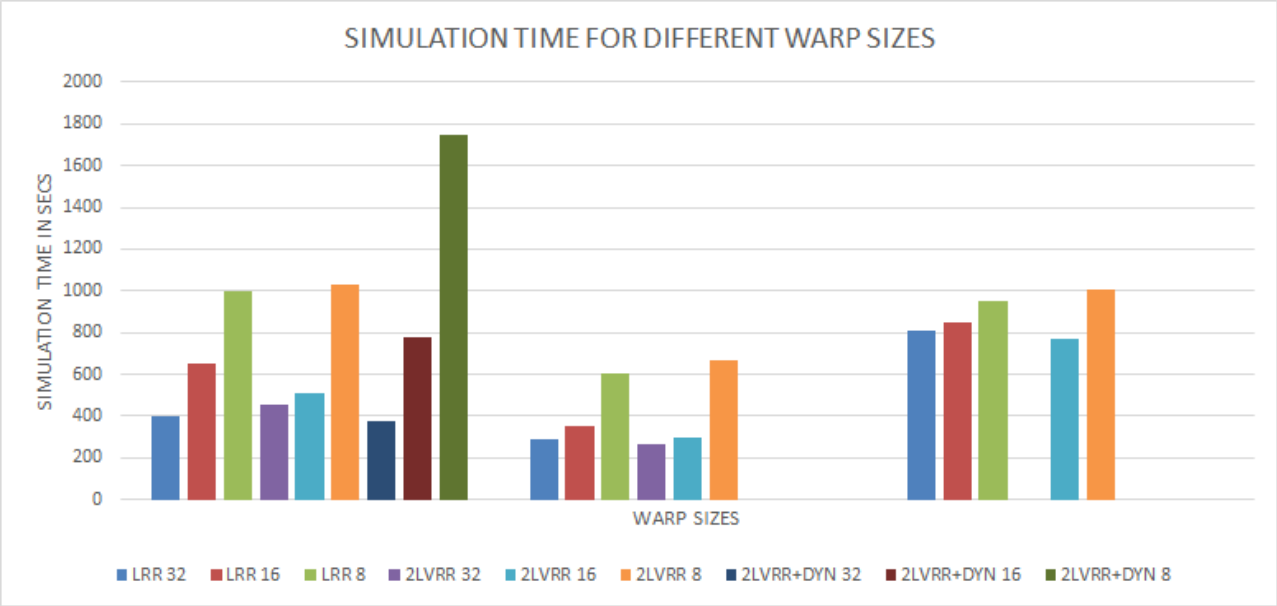
- In “abstract_hardware_model.h”, set MAX_WARP_SIZE (default is 32)
- In “shader.h”, set WARP_PER_CTA_MAX (default is 48)
- In “gpgpusim.config” (the one copied next to the benchmarks), set the “shader_core_pipeline”: the first part before the colon is the number of threads per block, and the second part after it is warp size (default is 1536:32).
    - (Only the second part was changed here)

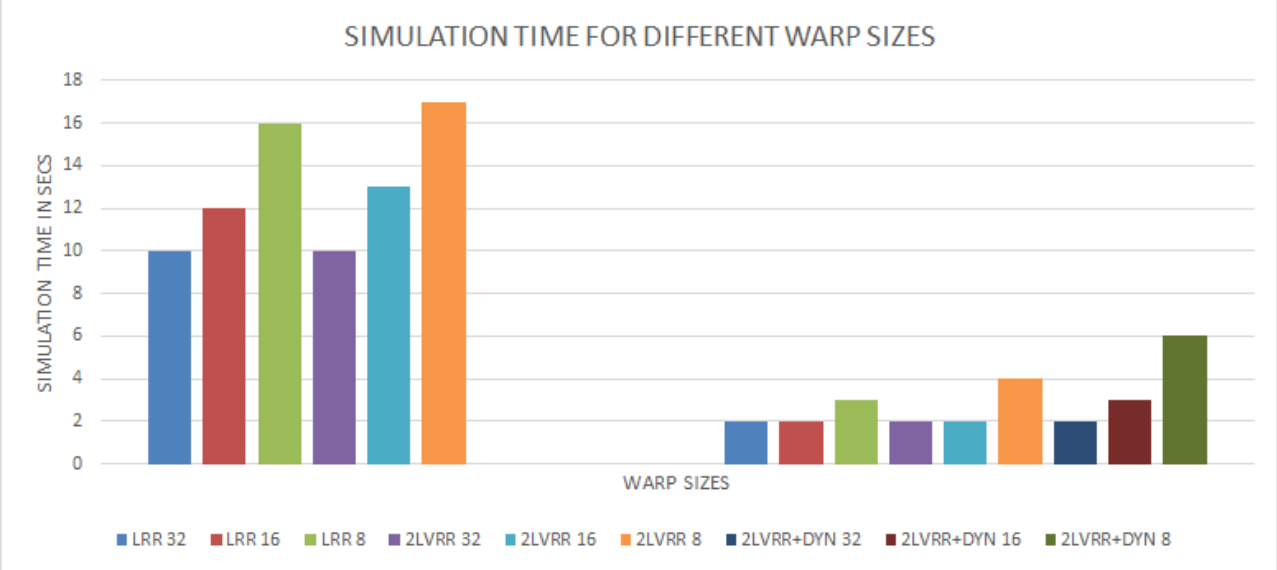

IPC FOR 32 WARP SIZE
IPC
700
600
500
400
300
200
100
0
BFS
BLK
HS
HIST
NN
LRR 32 21.17 575.67 512.6 153.23 35.02
2LVRR 32 21.25 569.56 520.04 153.23 0
2LVRR+DYN 32 0 569.56 0 153.23 0
WARP SIZES
LRR 32
2LVRR 32
2LVRR+DYN 32


IPC FOR DIFFERENT WARP SIZES
700
600
500
400
300
200
100
0
32
16
8
32
16
8
32
16
8
LRR
2LVRR
2LVRR+DYN
BLK
HS
HIST


IPC FOR DIFFERENT WARP SIZES
180
160
140
120
100
80
60
40
20
0
32
16
8
32
16
8
32
16
8
LRR
2LVRR
2LVRR+DYN
BFS
HIST


SIMULATION TIME FOR 32 WARP SIZE
900
800
700
600
500
400
300
200
100
0
BFS
BLK
HS
HIST
NN
LRR
2LVRR
2LVRR+DYN





Conclusion

In this project, we have used dynamic warps and two-level round robin scheduling. There are still many improvement possibilities in warp scheduling which can positively affect performance and power consumption, and we look forward to exploring them more in the near future.

Acknowledgement

The authors would like to thank Prof. Daniel Wong for sharing his most valuable experience with us. We would also like to express our gratitude to Prof. Abu-Ghazaleh for his kind support and guidance.

References

[1] V. Narasiman et al., "Improving GPU Performance via Large Warps and Two-Level Warp Scheduling," Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture. ACM, 2011.

[2] A. Bakhoda, G. Yuan, W. W. L. Fung, H. Wong and T. M. Aamodt, "Analyzing CUDA Workloads Using a Detailed GPU Simulator", IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Boston, MA, April 19-21, 2009.

[3] Y. Yu, W. Xiao, X. He, H. Guo, Y. Wang and X. Chen, "A Stall Aware Warp Scheduling For Dynamically Optimizing Thread-Level Parallelism In GPGPUs," ICS 2015 Proceedings of the 29th ACM on International Conference on Supercomputing:15-24.

[4] A. Jog et al., "OWL: cooperative thread array aware scheduling techniques for improving GPGPU performance," ACM SIGARCH Computer Architecture News 41.1 (2013): 395-406.

[5] S. Y. Lee and C. J. Wu, "CAWS: criticality-aware warp scheduling for GPGPU workloads," Proceedings of the 23rd international conference on Parallel architectures and compilation. ACM, 2014.

[6] C. Li, S. L. Song, H. Dai, A. Sidelnik, S. K. S. Hari, and H. Zhou, "Locality-Driven Dynamic GPU Cache Bypassing" (2015).

[7] T. G. Rogers et al., "A Variable Warp Size Architecture".

[8] M. Mao, J. Hu and H. Li, "VWS: A Versatile Warp Scheduler for Exploring Diverse Cache Localities of GPGPU Applications."