Implementation of Phase Aware Warp Scheduling Policy

Saurabh Labde,
Shrinath Cheriyana
Sanika Sabnis

Research Paper:

Phase Aware Warp Scheduling: Mitigating Effects of Phase Behaviour in GPGPU Applications

Mihir Awatramani, Xian Zhu, Joseph Zambreno and Diane Rover

ABSTRACT:

Graphics Processing Units (GPUs) have massive computational power and hence have become a popular platform for executing parallel applications. GPU models run the computations on thousands of threads parallelly. GPUs work on the principle of SIMD execution i.e. parallelism at the granularity of warps. The computational resources of GPU are not fully exploited to achieve the desired performance. Warp Scheduling plays an important role to use the resources effectively as it helps to hide the memory latencies and utilizes memory bandwidth efficiently.

There are two common warp scheduling policies widely used: Round Robin (RR) and Greedy then Oldest (GTO). But these policies work best for different applications. In this paper, the authors have proposed a warp scheduling policy that performs closer to the best of two policies. This report will explain the implementation of the proposed Phase Aware Warp Scheduling policy and demonstrate the experimental results observed.

INTRODUCTION:

Graphics Processing Units (GPU) have become popular as they exhibit high degree of data level parallelism. They are used as accelerators for not only graphics but also for general purpose applications. To achieve high computational throughputs, GPUs dedicate a large portion of the die area to functional units unlike CPUs. But due to this, GPU compromises on the hardware units used for hiding latency in CPUs. So, to hide these memory latencies GPU performs massive multithreading. Each core executes several threads parallelly in groups. These groups known as warps are scheduled by a hardware scheduler depending on the policy that is being used. So, warp scheduling plays a vital role to hide the latencies as it overlaps the latency of warp which is waiting on a long latency operation.

The two widely used warp scheduling policies are Round Robin (RR) and Greedy-Then-Oldest (GTO). RR scheduling policy gives equal priority to every warp in a round robin manner. While, GTO policy gives priority to the earlier launched warp. The performance of these scheduling policy is applications specific. So, there is no universal policy which gives a better performance for all types of applications. So, the authors of this paper have come up with a novel scheduling policy called as Phase Aware Warp Scheduling (PAWS) policy which overcomes the disadvantage of both the policies.

Warp Schedulers check which warp is ready and schedules the warp for execution. The warp scheduling logic needs to make a choice which warp to send if there are multiple warps that are ready. The decision is made according to the scheduling policy. The authors in this paper have implemented scheduling policy for both Single level and Two-level schedulers. In a Single level Scheduler, a single queue is maintained which keeps track of all the active warps in the core. The scheduling logic arbitrates among all the warps inside this queue and selects the suitable warp for execution. But, nowadays there are typically 48-64 active warps running on one SIMT core. It becomes time and energy consuming to search the entire warp queue to find the warp. Hence, a new two-level scheduler design is implemented to reduce arbitration time and power. Due to increasing number of warps per SIMT core, a hierarchy of warp queue is implemented to keep the number of active warps low. This makes arbitration logic simpler and time efficient. An additional structure “ready queue” is added which contains a smaller subset of warps (6-8). Group of warps inside the ready queue is known as a “Fetch Group”. Warps inside a fetch group are executed until they stall at a long latency instruction. If there is a long latency stall, then that warp is put back to the larger warp queue and another ready warp is brought into the ready queue.

RELATED WORK:

I. General Warp Scheduling Techniques

The applications in which warps have uniform latency, a fairness-based warp and DRAM access scheduling approach works best [4]. In hierarchical warp scheduler which forms the baseline for this paper, queue size of 6 gives less than 1% performance loss in comparison to a single level warp scheduler and the two-level policy results in a better overlap of memory latency and computation [2][3]. The shortest phase lengths are combined with previous phases by adding a priority shift instruction which makes priority selection policy as greedy until all short phases are completed [5]

II. Scheduling Techniques to Mitigate Warp Divergence

Dynamic Warp Formation(DWP) creates new warps from threads that fall on the same program path after divergence point [6]. To reduce the synchronization overhead due to warps from different thread block forming a warp, Thread Block Compaction (TBC) was proposed which adds threads block affinity to DWP as it creates new warps from diverged warps of the same thread block [7] [9].

Dynamic Warp Subdivision (DWS) divides a warp into splits on branch divergence and memory divergence. It improves latency hiding when a warp is split and encounters a cache miss by creating a warp split with threads that hit [8] [9].

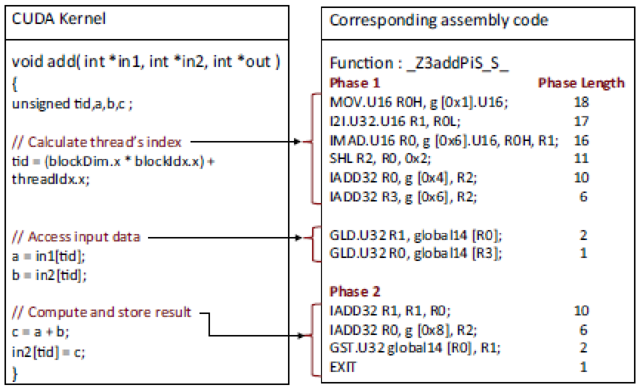
III. Thread Throttling

Performance of highly multithreaded architectures increases initially due to Thread Level Parallelism (TLP) and then starts to decrease once the total working set of threads is more than the cache size. [10] [11] [12] [13] [15]

Monitoring cache lines to detect warps that have less intra-warp locality because of other warps evicting data and not selecting these to reduce number of active warps [13]. The number of threads is reduced if the time spent waiting for memory is more than threshold [11]. Optimal amount of TLP is found by launching maximum number of warps and then using greedy scheduling policy [15]. Optimal thread block count is obtained by launching half the maximum number of warps and then use history information of previous block counts [12]. The warp execution is throttled by assigning a time slice to each warp proportional to its execution time with RR scheduler, which reduces tail effect by giving larger time slice to longer running warps [14].

DESIGN:

As discussed earlier there is no consensus regarding which scheduling policy (Round Robin or Greedy Then Oldest) works best for all the applications. The authors have suggested that the scheduling policy should be decided based on the characteristics of the instructions in different regions of the applications. The authors identify these regions as phases and define a phase as *"A set of consecutive instructions such that, no instruction in the current phase has an input operand that is produced by a long latency instruction from the current phase"*. Consider the example



CUDA Kernel

```
void add( int *in1, int *in2, int *out )
{
unsigned tid,a,b,c ;

// Calculate thread's index
tid = (blockDim.x * blockIdx.x) +
threadIdx.x;

// Access input data
a = in1[tid];
b = in2[tid];

// Compute and store result
c = a + b;
in2[tid] = c;
}
```

Corresponding assembly code

```
Function : _Z3addPiS_S_
Phase 1                                  Phase Length
MOV.U16 R0H, g [0x1].U16;                18
I2I.U32.U16 R1, R0L;                     17
IMAD.U16 R0, g [0x6].U16, R0H, R1;       16
SHL R2, R0, 0x2;                         11
IADD32 R0, g [0x4], R2;                  10
IADD32 R3, g [0x6], R2;                   6

GLD.U32 R1, global14 [R0];                2
GLD.U32 R0, global14 [R3];                1

Phase 2
IADD32 R1, R1, R0;                       10
IADD32 R0, g [0x8], R2;                   6
GST.U32 global14 [R0], R1;                2
EXIT                                      1
```

Figure 1: CUDA Kernel for vector addition and its corresponding assembly code (right) showing phase length and phase distance.

in Figure 1. Memory load instructions (GLD) are long latency instructions. As observed from the assembly code (Figure 1: Right) the registers R1 and R0 are destination operands of the GLD instructions and the first instruction that uses any of these registers (R0 or R1) as its source operand will mark the start of the new phase. However, it should be noted that it is very common to have multiple instructions in a given phase that depend on long latency instructions of previous phase. A new phase is only created if the instruction depends on the long latency instruction of the current phase. The authors define two more terms which are related to a phase. These are phase length and phase distance. The phase length is calculated by summing the instruction latencies from the last instruction on the phase to the first instruction in the phase. The phase distance for an instruction is defined as the approximate number of cycles required to reach the end of phase for that instruction.

For a two-level warp scheduler, a warp is popped out from the ready queue when it witnesses a long latency instruction. Since, long latency instructions occur at the boundary of a phase it can be concluded that in a kernel which is divided into phases, the warps proceed through the kernel at the granularity of phases. Thus, when a warp reaches the end of its current phase it is switched to the larger pool of warps and thus, it can be inferred that a warp maintains the its priority

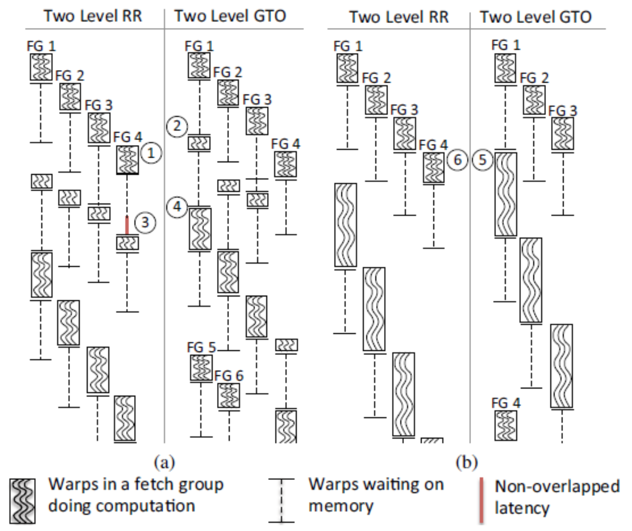


Figure 2: Effect of phase length on warp scheduling.

till it reaches the end of its phase. Thus, the phase length factor has a significant impact on the scheduling policy.

Consider the example shown in Figure 2. The Figure shows two applications which are scheduled using RR and GTO scheduling policies. The kernel for the application (a) is divided into phases in the following manner: medium phase followed by short phase and long phase while the kernel of application (b) is divided into medium phase followed by long phase followed by a short phase. Four fetch groups must be scheduled, and the illustration assumes that all the warps in a fetch group reaches the end of phase simultaneously. It is observed that application (a) performs better when scheduled using the GTO policy while application (b) performs better when scheduled with RR policy. In general, GTO performs better for applications that have short phases in the middle of the kernel and RR performs better for applications that have long phases in the middle of the kernel.

One key observation that the authors make from the above example is that an application performed better when the scheduling policy (RR or GTO) gave higher priority to the FG group which had the shortest phase length for its next phase. This observation is the foundation for the scheduling policy proposed by the authors.

The authors' proposed Phase Aware Warp Scheduling (PAWS) Policy states that *"Adverse effects of RR or GTO policy can be mitigated by always choosing the warp that has the shortest length for its next phases"*. The PAWS policy states that warps should be scheduled based on the phase lengths. The warp which has the shortest phase length for its next phase should be assigned a higher priority.

IMPLEMENTATION

The simulator uses the Tesla C2050 with the following specifications:

Chip Configuration	
Number of cores	14
Core frequency	1150 MHz
DRAM clock frequency	1500 MHz
Peak SP / DP floating point throughput	1000/515 GFLOPs
Peak DRAM bandwidth	144 GB/sec
Core configuration	
Maximum thread blocks per core	8
Maximum warps supported per core	48
Execution units per core	16
Scheduler configuration	
Warp schedulers per core	2
Instruction dispatch throughput per scheduler	1
Ready warps queue size	6

The following section explains the changes made to the GPGPU simulator to incorporate Phase Aware Warp Scheduling. The implementation is divided into two sections which are calculation of the phase information and the phase aware warp scheduler.

I. PHASE CALCULATION:

Files Modified:

"ptx_ir.h" and *"abstract_hardware_model.h"*

The first step in phase calculation is to check which instructions have source operands that depend on long latency instructions. Code shown in Figure 4 traverses the instructions in a kernel once to divide the kernel code into phases. To achieve this, instructions stored in the *m_instr_mem* array are checked one by one to detect LOAD instructions. If the instruction is a LOAD, then its destination register is stored in a separate vector called *long_op*. In the same iteration the source operands of the instructions are also checked. If any of the source operand registers matches with the registers stored in the *long_op* vector, then the phase number is incremented, and the current instruction is marked as the start of phase and the previous instruction is marked as the end of phase and the *long_op* vector is cleared and made ready for next phase. After this the instructions are traversed from bottom to top to calculate the phase distance (Figure 5).

```
int phase_id;
int get_phase_id() const { return phase_id; }
void set_phase_id(int p_id) { phase_id = p_id; }
bool end_of_phase;
bool get_end_of_phase() const{ return end_of_phase; }
void mark_end_of_phase() { end_of_phase = true; }
void mark_start_of_phase() { end_of_phase = false; }

int phase_length;
int phase_distance;
int total_phases;
int get_phase_distance() const { return phase_length; }
void set_phase_distance (int p_length) { phase_length = p_length; }
```

Figure 3: Code added to *"abstract_hardware_model.h"*.

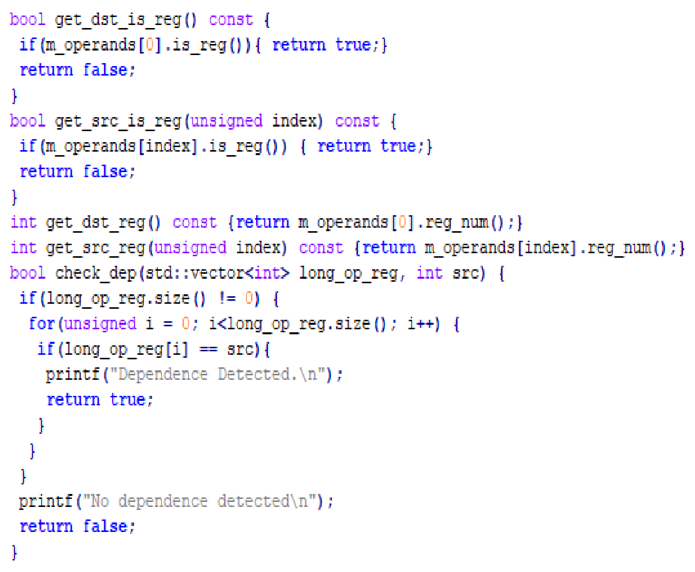
```
if (phase_calculated == false)    {
 for(unsigned i = 0; i<m_instr_mem_size; i++)   {
  if(m_instr_mem[i] != NULL)   {
   instr = m_instr_mem[i];
   num_of_operands =instr->get_num_operands();

   if(num_of_operands != 0) {
    d_is_reg = instr->get_dst_is_reg();
    if (d_is_reg)    {
     destination_reg = instr->get_dst_reg();
     if(instr->op == LOAD_OP && instr->memory_op == memory_load) {
      long_op.push_back(destination_reg);
     }
    }
    for(unsigned j = 1; j<num_of_operands; j++) {
     s_is_reg = instr->get_src_is_reg(j);
     if(s_is_reg){
      source_reg = instr->get_src_reg(j);
      dependence_flag = instr->check_dep(long_op,source_reg);
      if(dependence_flag == true) {
       phase_id++;
       prev_instr->mark_end_of_phase();
       instr->mark_start_of_phase();
       long_op.clear();
       break;
      }
     }
    }
   }
   instr->set_phase_id(phase_id);
  }
  prev_instr = instr;
 }
```

Figure 4: Code section responsible for dividing the kernel code into phases.

```
for(int k = m_instr_mem_size - 1; k>= 0; k--) {
 if(m_instr_mem[k] != NULL){
  instr = m_instr_mem[k];
   if (instr->get_end_of_phase()) { phase_dis = 0;}
   phase_dis += instr->latency;
   instr->set_phase_distance(phase_dis);
 }
}
```

Figure 5: Code section responsible for calculating phase distance.



```
bool get_dst_is_reg() const {
 if(m_operands[0].is_reg()){ return true;}
 return false;
}
bool get_src_is_reg(unsigned index) const {
 if(m_operands[index].is_reg()) { return true;}
 return false;
}
int get_dst_reg() const {return m_operands[0].reg_num();}
int get_src_reg(unsigned index) const {return m_operands[index].reg_num();}
bool check_dep(std::vector<int> long_op_reg, int src) {
 if(long_op_reg.size() != 0) {
  for(unsigned i = 0; i<long_op_reg.size(); i++) {
   if(long_op_reg[i] == src){
    printf("Dependence Detected.\n");
    return true;
   }
  }
 }
 printf("No dependence detected\n");
 return false;
}
```

Figure 6: Function for checking dependencies.

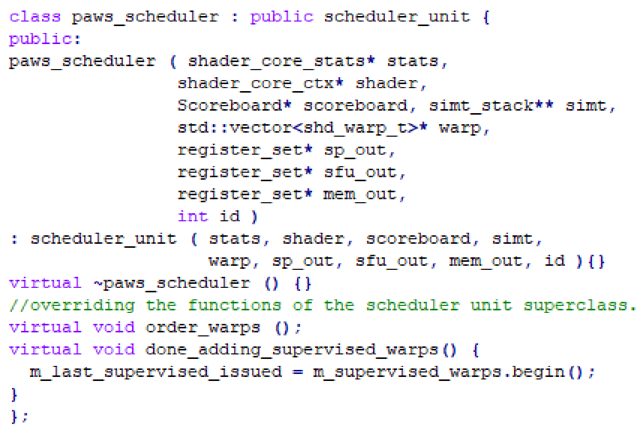
II. PHASE AWARE WARP SCHEDULUER

Files Modified:

"shader.h" and *"shader.cc"*

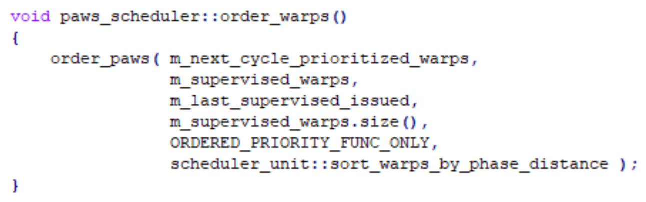
This section deals with the implementation of the phase aware warp scheduler using the phase information calculated in the previous section. The priority assigning logic of the phase aware warp scheduler assigns higher priority to the warps that have the shortest phase distance for its next phase.

A new subclass called *paws_scheduler* of the superclass *scheduler_unit* is created. This class contains methods *order_warps* which override the method of the superclass *schedueler_unit*.



```
class paws_scheduler : public scheduler_unit {
public:
paws_scheduler ( shader_core_stats* stats,
                shader_core_ctx* shader,
                Scoreboard* scoreboard, simt_stack** simt,
                std::vector<shd_warp_t>* warp,
                register_set* sp_out,
                register_set* sfu_out,
                register_set* mem_out,
                int id )
: scheduler_unit ( stats, shader, scoreboard, simt,
                   warp, sp_out, sfu_out, mem_out, id ){}
virtual ~paws_scheduler () {}
//overriding the functions of the scheduler unit superclass.
virtual void order_warps ();
virtual void done_adding_supervised_warps() {
  m_last_supervised_issued = m_supervised_warps.begin();
}
};
```

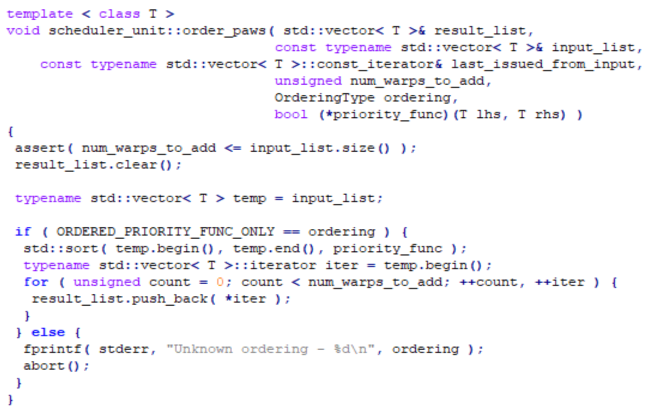
Figure 7: Subclass *"paws_scheduler"*.



```
void paws_scheduler::order_warps()
{
    order_paws( m_next_cycle_prioritized_warps,
                m_supervised_warps,
                m_last_supervised_issued,
                m_supervised_warps.size(),
                ORDERED_PRIORITY_FUNC_ONLY,
                scheduler_unit::sort_warps_by_phase_distance );
}
```

Figure 8: Function that overrides the *order_warps* function of *scheduler_unit*.

The *order_warps* function of the subclass *paws_scheduler* calls the function *order_paws* which decides the priority of warps based on the phase distance and interchange the warp orders.



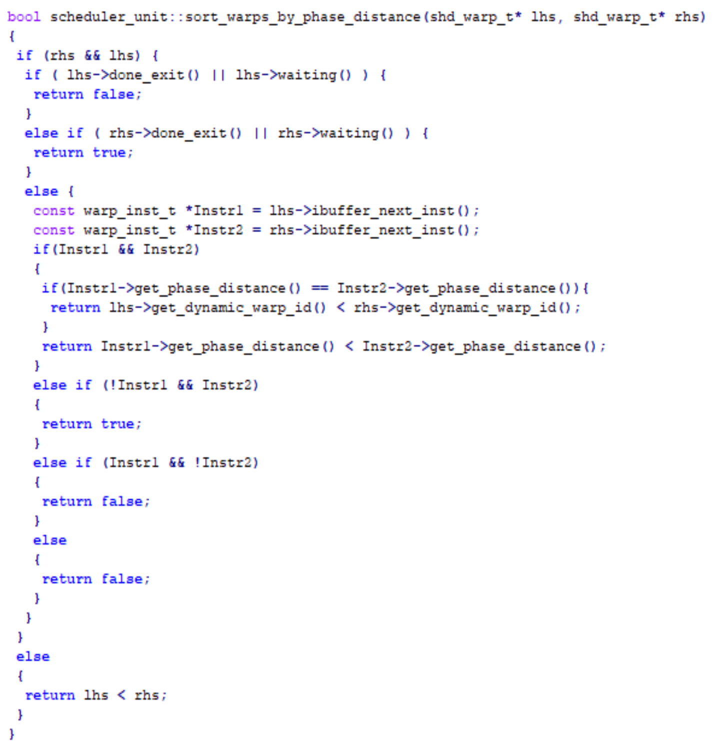
```
template < class T >
void scheduler_unit::order_paws( std::vector< T >& result_list,
                                const typename std::vector< T >& input_list,
    const typename std::vector< T >::const_iterator& last_issued_from_input,
                                unsigned num_warps_to_add,
                                OrderingType ordering,
                                bool (*priority_func)(T lhs, T rhs) )
{
 assert( num_warps_to_add <= input_list.size() );
 result_list.clear();

 typename std::vector< T > temp = input_list;

 if ( ORDERED_PRIORITY_FUNC_ONLY == ordering ) {
  std::sort( temp.begin(), temp.end(), priority_func );
  typename std::vector< T >::iterator iter = temp.begin();
  for ( unsigned count = 0; count < num_warps_to_add; ++count, ++iter ) {
   result_list.push_back( *iter );
  }
 } else {
  fprintf( stderr, "Unknown ordering - %d\n", ordering );
  abort();
 }
}
```

Figure 9: Function *"order_paws"*.

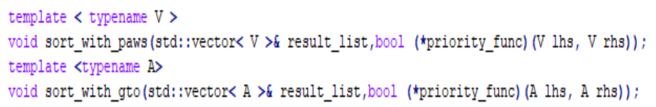
The *order_paws* function uses a Boolean function called *sort_warps_by_phase_distance* to sort the warps according to the phase distance. The *sort_warps_by_phase_distance* function compares the phase distances of instructions in different warps and sorts the warps accordingly giving higher priority to warp which has the instruction with the shortest phase distance. In the case when phase distances are equal, oldest warp is given priority. The output of this function is provided to the sort function of *order_paws* which allots the priority to the warps based on phase distance.



```
bool scheduler_unit::sort_warps_by_phase_distance(shd_warp_t* lhs, shd_warp_t* rhs)
{
 if (rhs && lhs) {
  if ( lhs->done_exit() || lhs->waiting() ) {
   return false;
  }
  else if ( rhs->done_exit() || rhs->waiting() ) {
   return true;
  }
  else {
   const warp_inst_t *Instr1 = lhs->ibuffer_next_inst();
   const warp_inst_t *Instr2 = rhs->ibuffer_next_inst();
   if(Instr1 && Instr2)
   {
    if(Instr1->get_phase_distance() == Instr2->get_phase_distance()){
     return lhs->get_dynamic_warp_id() < rhs->get_dynamic_warp_id();
    }
    return Instr1->get_phase_distance() < Instr2->get_phase_distance();
   }
   else if (!Instr1 && Instr2)
   {
    return true;
   }
   else if (Instr1 && !Instr2)
   {
    return false;
   }
   else
   {
    return false;
   }
  }
 }
 else
 {
  return lhs < rhs;
 }
}
```

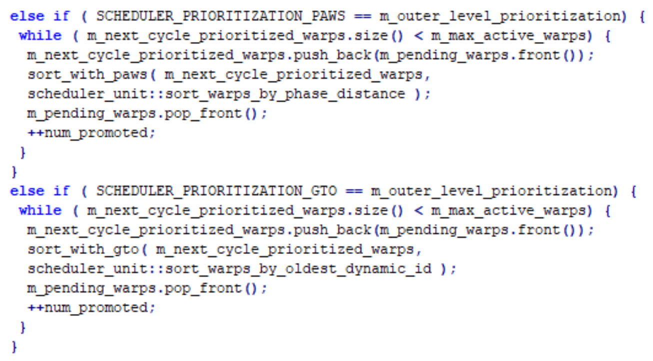
Figure 10: Function *"sort_warps_by_phase_distance"*.

For two-level warp scheduler the current GPGPU simulator consisted only RR. Two more options were added to the existing code to allow the two-level scheduler to be scheduled with GTO and PAWS policy.



```
template < typename V >
void sort_with_paws(std::vector< V >& result_list,bool (*priority_func)(V lhs, V rhs));
template <typename A>
void sort_with_gto(std::vector< A >& result_list,bool (*priority_func)(A lhs, A rhs));
```

Figure 11: Code added in *"shader.h"*.



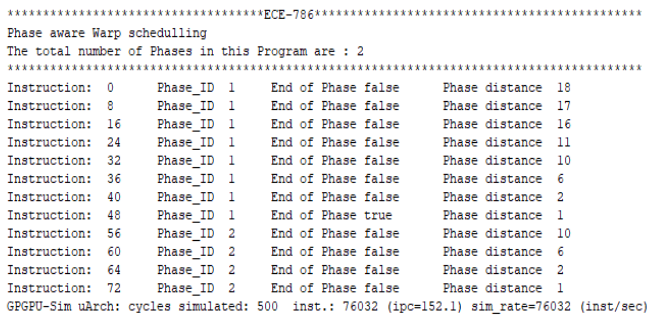
```
else if ( SCHEDULER_PRIORITIZATION_PAWS == m_outer_level_prioritization) {
 while ( m_next_cycle_prioritized_warps.size() < m_max_active_warps) {
  m_next_cycle_prioritized_warps.push_back(m_pending_warps.front());
  sort_with_paws( m_next_cycle_prioritized_warps,
  scheduler_unit::sort_warps_by_phase_distance );
  m_pending_warps.pop_front();
  ++num_promoted;
 }
}
else if ( SCHEDULER_PRIORITIZATION_GTO == m_outer_level_prioritization) {
 while ( m_next_cycle_prioritized_warps.size() < m_max_active_warps) {
  m_next_cycle_prioritized_warps.push_back(m_pending_warps.front());
  sort_with_gto( m_next_cycle_prioritized_warps,
  scheduler_unit::sort_warps_by_oldest_dynamic_id );
  m_pending_warps.pop_front();
  ++num_promoted;
 }
}
```

Figure 12: Additional options for sorting outer queue for two level schedulers.

RESULTS

Kernels from the Rodinia Benchmark Suites were used to evaluate the performance of the implementation. The graphs shown below depict the performance of RR, GTO and PAWS scheduling policy for the respective kernels.

For the vectorAdd kernel the Figure 10 shows the phases calculated and the phase distance associated with each instruction.

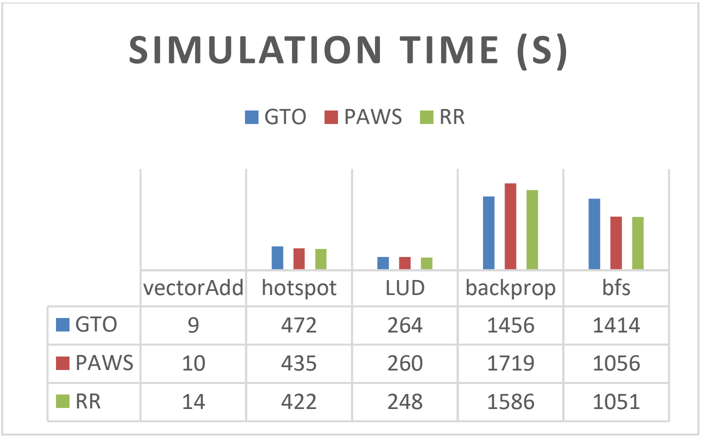


```
**********************************ECE-786********************************************
Phase aware Warp schedulling
The total number of Phases in this Program are : 2
*************************************************************************************
Instruction:  0      Phase_ID  1     End of Phase false      Phase distance  18
Instruction:  8      Phase_ID  1     End of Phase false      Phase distance  17
Instruction:  16     Phase_ID  1     End of Phase false      Phase distance  16
Instruction:  24     Phase_ID  1     End of Phase false      Phase distance  11
Instruction:  32     Phase_ID  1     End of Phase false      Phase distance  10
Instruction:  36     Phase_ID  1     End of Phase false      Phase distance  6
Instruction:  40     Phase_ID  1     End of Phase false      Phase distance  2
Instruction:  48     Phase_ID  1     End of Phase true       Phase distance  1
Instruction:  56     Phase_ID  2     End of Phase false      Phase distance  10
Instruction:  60     Phase_ID  2     End of Phase false      Phase distance  6
Instruction:  64     Phase_ID  2     End of Phase false      Phase distance  2
Instruction:  72     Phase_ID  2     End of Phase false      Phase distance  1
GPGPU-Sim uArch: cycles simulated: 500  inst.: 76032 (ipc=152.1) sim_rate=76032 (inst/sec)
```

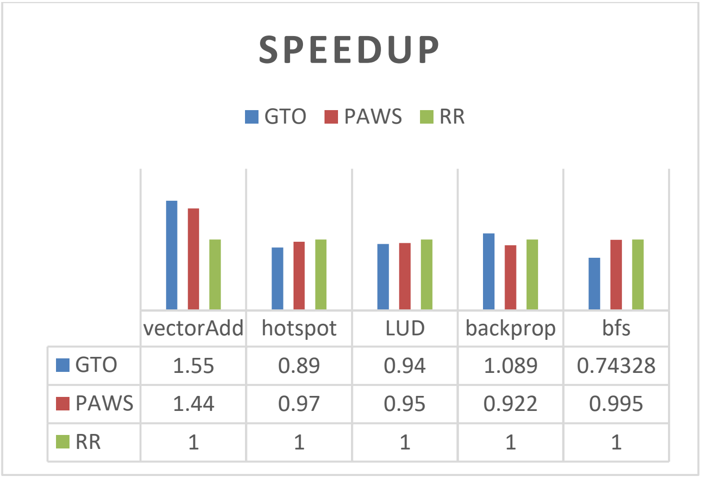
Figure 13: Phase calculation for vectorAdd kernel.

Single level warp scheduler:

In vectorAdd and backprop kernels we observed that GTO outperformed the RR policy. The implemented PAWS policy is seen to outperform RR in vectorAdd kernel and performs closely to GTO. But in backprop kernel RR policy outperforms PAWS policy. In hotspot, bfs and lud kernels we observed that RR outperforms GTO and PAWS follows closely to the RR policy.



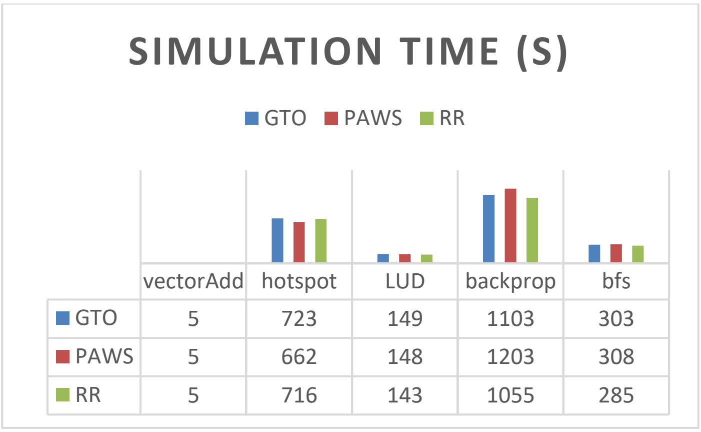
	vectorAdd	hotspot	LUD	backprop	bfs
GTO	9	472	264	1456	1414
PAWS	10	435	260	1719	1056
RR	14	422	248	1586	1051



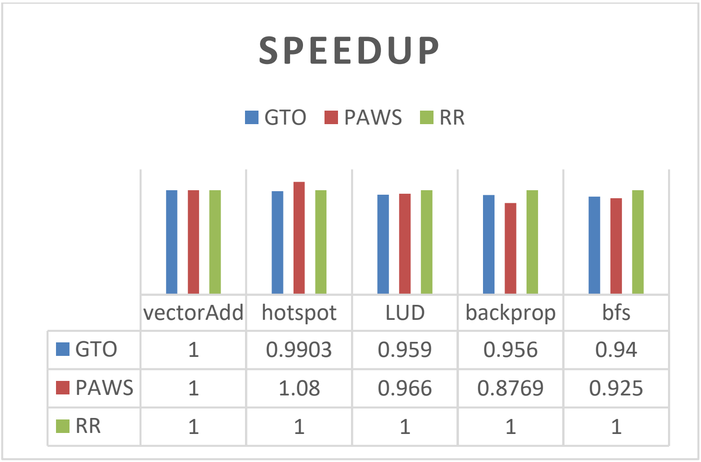
	vectorAdd	hotspot	LUD	backprop	bfs
GTO	1.55	0.89	0.94	1.089	0.74328
PAWS	1.44	0.97	0.95	0.922	0.995
RR	1	1	1	1	1

Two-level warp scheduler:

The trends are similar to Single level scheduler.



	vectorAdd	hotspot	LUD	backprop	bfs
GTO	5	723	149	1103	303
PAWS	5	662	148	1203	308
RR	5	716	143	1055	285



	vectorAdd	hotspot	LUD	backprop	bfs
GTO	1	0.9903	0.959	0.956	0.94
PAWS	1	1.08	0.966	0.8769	0.925
RR	1	1	1	1	1

CONCLUSION

It can be observed from the results that the Phase Aware Warp Scheduling policy performs close to the scheduling policy (RR or GTO) that works best for an application. This justifies the authors' claim that PAWS can be used as a common scheduling policy for all the applications.

REFERENCES

1. Mihir Awatramani, Xian Zhu, Joseph Zambreno and Diane T. Rover, "Phase Aware Warp Scheduling: Mitigating Effects of Phase Behavior in GPGPU Applications", *Proc. of Int. Conf. on Parallel Architectures and Compilation Techniques*, pp. 1-12, October 2015
2. M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler,W. J. Dally, E. Lindholm, and K. Skadron, "A Hierarchical Thread Scheduler and Register File for Energy-Efficient Throughput Processors," *ACM Trans. Computer Syst.*, 2012.
3. V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt, "Improving GPU Performance via Large Warps and Two-Level Warp Scheduling," in *Proc. of the 44th Annual IEEE/ACM Int. Symp. on Microarchitecture*, 2011.
4. N. B. Lakshminarayana and H. Kim, "Effect of Instruction Fetch and Memory Scheduling on GPU Performance," in *Workshop on Language, Compiler and Architecture Support for GPGPU*, 2010.
5. J. Chen, X. Tao, Z. Yang, J.-K. Peir, X. Li, and S.-L.Lu, "Guided Region-Based GPU Scheduling: Utilizing Multi-thread Parallelism to Hide Memory Latency," in *IEEE 27th Int. Symp. on Parallel & Distributed Processing*.
6. W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic Warp Formation: Efficient MIMD Control Flow on SIMD Graphics Hardware," *ACM Trans. Archit. Code Optim.*, 2009.
7. W. W. L. Fung and T. M. Aamodt, "Thread Block Compaction for Efficient SIMT Control Flow," in *Proc. of the 2011 IEEE Int. Symp. on High Performance Computer Architecture*, 2011.
8. J. Meng, D. Tarjan, and K. Skadron, "Dynamic Warp Subdivision for Integrated Branch and Memory Divergence Tolerance," in *Proc. of the 37th Annual Int. Symp. on Computer Architecture*, 2010.
9. M. Steffen and J. Zambreno, "Improving SIMT Efficiency of Global Rendering Algorithms with Architectural Support for Dynamic Micro-Kernels," in *Proc. Of the 43rd Annual IEEE/ACM Int. Symp. on Microarchitecture*, 2010.
10. Z. Guz, E. Bolotin, I. Keidar, A. Kolodny, A. Mendelson, and U. C. Weiser, "Many-core vs. Many-Thread Machines: Stay Away From the Valley," *IEEE Computer Architecture Letters*, 2009.
11. O. Kayiran, A. Jog, M. T. Kandemir, and C. R. Das, "Neither More nor Less: Optimizing Thread-level Parallelism for GPGPUs," in *Proc. of the 22nd Int. Conf. on Parallel Architectures and Compilation Techniques*, 2013.
12. M. Awatramani, D. Rover, and J. Zambreno, "Perf-Sat: Runtime Detection of Performance Saturation for GPGPU Applications," in *Proc. of the Int. Workshop on Scheduling and Resource Management for Parallel and Distributed Systems (SRMPDS)*, 2014.
13. T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-Conscious Wavefront Scheduling," in *Proc. of the 45th Annual IEEE/ACM Int. Symp. on Microarchitecture*, 2012.
14. S. Lee, and C. Wu, "CAWS: Criticality Aware Warp Scheduling for GPGPU Workloads," in *Proc. of the 23rd Int. Conf. on Parallel Architectures and Compilation Techniques*, 2014.
15. M. Lee, S. Song, J. Moon, J. Kim, W. Seo, Y. Cho, and S. Ryu, "Improving GPGPU Resource Utilization Through Alternative Thread Block Scheduling," in *IEEE 20th Int. Symp. on High Performance Computer Architecture (HPCA)*, 2014.
16. GPGPU-Sim 3.1.1. Manual, *Website:* http://people.cs.pitt.edu/~yongli/notes/gpgpu/GPGPUSIMNotes.html
17. NVIDIA CUDA C Programming Guide, *Website:* http://gpgpu-sim.org/manual/index.php/Main_Page