Taming Warp Divergence

Jayvant Anantpur R. Govindarajan
Supercomputer Education and Research Centre
Indian Institute of Science, Bangalore, India
jayvant.anantpur@gmail.com govind@serc.iisc.in

Abstract

Graphics Processing Units (GPUs) are designed to exploit large amount of parallelism. However, warp-level divergence occurring due to different amounts of work, memory access latency experienced, etc., results in warps of a thread block (TB) finishing kernel execution at different points in time. This, in effect, reduces utilization of resources of SMs and hence performance of the GPU.

We propose a simple and elegant technique to eliminate the waiting time of warps at the end of kernel execution and improve performance. The proposed technique uses the idea of persistent threads to define virtual thread blocks and virtual warps. This enables the virtual warp finishing earlier to initiate the execution of another warp from a subsequent thread block, avoiding the unnecessary waiting for sibling warps and the resulting resource underutilization. Further, this technique enables us to design a warp scheduling algorithm that is aware of the progress made by the virtual thread blocks and virtual warps, and uses this knowledge to prioritise warps effectively. The proposed approach is implemented using a simple source-to-source transformation and minimal hardware support. Evaluation of the proposed approach on a diverse set of kernels on the GPGPU-Sim simulator reveals a geometric mean improvement of 1.06x over the baseline architecture that uses the Greedy Then Old (GTO) warp scheduler and 1.09x over the Loose Round Robin (LRR) warp scheduler.

Keywords GPU, Warp Scheduling, Divergence

1. Introduction

Increasing compute power of Graphics Processing Units (GPU) and availability of programming models such as CUDA [7] and OpenCL [25] have enabled the use of GPUs to speed-up not only data parallel code but also various irregular applications. A typical GPU consists of a few tens of Streaming Multiprocessors (SM)[1]. Each SM consists of a number of simple in-order cores, amounting to several hundreds or even thousands of SIMD cores in a GPU.

[1] We use the terminology of NVIDIA GPUs and Compute Unified Device Architecture (CUDA) throughout the paper.

Program execution happens in a GPU in the form of a kernel launch with a hierarchical execution configuration, called *grid*, which consists of a number of thread blocks (TBs), and each TB block consisting of a number of threads. Threads are allocated and deallocated to the SM at the granularity of a TB [9], whereas thread scheduling and execution happen at the granularity of a group of consecuting threads (typically 32 threads), called *warp*. This is done by the warp scheduler in an SM which has the ability to select one warp every cycle from a group of ready warps.

When a kernel is invoked, the TB scheduler on the GPU allocates as many TBs to each SM, as allowed by the resource constraints, and the remaining TBs are assigned, one by one, to an SM as and when it has enough free resources to accommodate a new TB. A TB typically consists of multiple warps and the warps of a TB may take different amounts of time to finish execution of the kernel code. This behavior, called warp-level divergence [36] or simply warp divergence, causes the warps finishing earlier to wait for their sibling warps. Only when all warps of a TB finish, its resources can be deallocated and a new TB can be assigned for execution.

It has been observed in [36] that warp-level divergence occurs due to differing (i) amount of work, (ii) memory latency experienced, or (iii) priority assigned by the warp scheduler policy. The policy of allocation and deallocation of resources at TB granularity and warp divergence, together, cause the hardware resources of SMs to remain unutilized when warps are waiting for their siblings, reducing the performance of the GPU. When warps are waiting for their siblings to finish execution, the number of warps available for scheduling goes down, reducing the ability of warp schedulers to effectively hide long execution latencies.

To improve the performance of applications exhibiting warp divergence, we propose a technique, called Virtual Thread Blocks and Warps, *VTW*, that uses the concept of persistent threads [12] to define virtual TBs and virtual Warps. VTW achieves this by modifying the kernel code to use parameterized thread and TB indices, and enclosing the kernel code in an iterative loop. Thus a virtual TB can execute the kernel code on behalf of multiple logical TBs – TBs of the grid as specified by the user – by setting the parame-

978-1-5090-4931-8/17$15.00

terized thread and TB indices appropriately. This enables a virtual warp finishing early to start execution of the kernel code again for a new logical TB, without waiting for its sibling warps to finish[2]. In other words, a virtual TB iterates over the kernel code multiple times. As the number of resident warps available for scheduling remains the same as the initial number of warps – as long as there are logical TBs remaining to be assigned – VTW retains the ability to mask latencies even in the presence of warp-level divergence. VTW requires simple ISA extensions to compute the next logical TB index and warp index to be assigned to each virtual warp when it starts a new iteration of the kernel code. While VTW can be made purely as a software scheme, without requiring ISA extensions, its performance diminishes due to the additional register requirements and the resulting reduction in TB residency.

The proposed VTW scheme facilitates several optimizations. First, warp schedulers can be made aware of the iteration numbers of virtual warps and virtual TBs, and can prioritise virtual warps effectively. We use this idea to propose a progress-aware warp scheduling method which further speeds up the application.

Use of iterative loop enables compiler optimizations such as loop independent code motion that moves instructions that are independent of thread and TB block indices outside the iterative loop. This eliminates duplicate execution of code that is independent of thread and TB block indices. Last, since a virtual TB can execute the kernel code for multiple logical TBs, users can control the number of virtual TBs to be launched by setting the number of TBs in the execution configuration, enabling the user to control the number of resources used by a kernel launch. This in turn facilitates efficient concurrent kernel execution.

The main contributions of the paper are:

- VTW, an elegant virtual TB and virtual warp mechanism, to tame warp-level divergence,
- Simple source-to-source transformation and ISA extension implementation of VTW,
- Design of a warp scheduling algorithm that is progress aware and effectively uses virtual warps to hide long execution latencies,
- Detailed evaluation on 51 kernels from Rodinia [5], Parboil [34], and GPGPU-SIM [4] benchmark suites. VTW achieves a geometric mean performance improvement of 1.06x over a baseline that uses GTO warp scheduler and 1.09x over LRR.

To the best of our knowledge, our work is the first to use the concept of virtual TBs and virtual warps to handle warp divergence problem.

[2] VTW handles kernels with shared memory accesses also.

2. Background

A typical GPU consists of a number of Streaming Multiprocessors (SM) and each SM contains tens to hundreds of simple in-order cores. For example, the Kepler GK110 GPU [16] consists of 15 SMs and each SM contains 192 CUDA cores, with each core containing a fully pipelined integer arithmetic logic unit and floating point unit. In addition, each SM also contains 32 Load/Store Units, 32 Special Function Units, 64 Double Precision units, a register file of 64K 4 byte registers, L1 cache and software managed shared memory. CUDA [7] and OpenCL [25] are two of the most commonly used GPU programming languages, which use Single Instruction Multiple Threads (SIMT) computation model.

Functions to be executed on a GPU are called kernels. A kernel is invoked with an execution configuration called grid, which specifies the number of threads per TB, and number of TBs, using a hierarchical structure which can be up to 3 dimensional. Using shared memory and barrier synchronization, threads of a TB can communicate and cooperate with each other. Thread blocks run independent of each other.

Each SM has a limited number of resources such as registers, shared memory, etc., and hence the number of TBs that can be allocated to an SM depends on the resources required by each thread and TB. For example, each SM on Kepler GK110 GPU can hold up to 16 TBs at a time. On a kernel invocation, a global work distribution engine (TB Scheduler) in the GPU assigns as many TBs to an SM as allowed by resource constraints and then assigns the remaining TBs one at a time as and when a previously assigned TB finishes.

Threads of a TB are further partitioned into groups of consecutive 32 threads, called warps in the CUDA terminology. Each SM contains one or more warp schedulers, each of which schedules one ready warp and issues the next instruction from the selected warp to the execution pipeline.

3. Motivation

As mentioned in Section 1, threads and GPU resources are allocated to and deallocated from SMs at the TB granularity. A TB finishes execution when all its warps complete their execution. However, the warps of a TB can take different amounts of time to finish their execution, resulting in warp divergence [36]. One reason for warp divergence is the amount of work done by warps of a TB may be different, due to input data dependent loops and branches. Another, is the global memory access latencies experienced by different warps may be very different due to cache misses, uncoalesced memory accesses, etc. One more reason is that the priorities assigned by warp schedulers can cause some warps to get more compute cycles than others.

To measure amount of divergence present among warps of a TB, we evaluated various kernels on the GPGPU-Sim simulator using LRR as the warp scheduler. We chose LRR due to its fairness in scheduling warps, and choosing a fair warp scheduler will highlight the inherent divergence in a

Table 1: Warp Divergence in Kernels

BM	Kernel	TBs	Res	Iters	Instruction Divergence				Cycle Divergence			
					50%+	25%+	10%+	5%+	50%+	25%+	10%+	5%+
P_lbm	performStreamColl	18000	105	172	4841	17999	17999	17999	17995	17999	18000	18000
R_hybridsort	mergepack	17408	90	194	367	618	846	936	3519	11537	17026	17360
R_bfs	Kernel	1954	45	44	525	755	1005	1015	1017	1018	1210	1695
R_bfs	Kernel2	1954	45	44	1	1	159	492	925	1016	1146	1621
R_hybridsort	mergeSortPass	695	75	10	694	694	695	695	679	685	694	694
I_BFS	Kernel	256	90	3	48	98	130	133	135	145	180	209
I_MUM	mummergpuKernel	196	75	3	1	1	21	176	140	183	196	196
I_NQU	solve_nqueen_cu	256	45	6	0	0	255	255	60	213	255	255
I_RAY	render	512	75	7	27	53	108	128	55	86	107	116
R_b+tree	findK	10000	90	112	0	0	0	9073	43	8642	9999	9999
I_NN	executeFirstLayer	168	105	2	168	168	168	168	1	106	168	168
R_b+tree	findRangeK	6000	90	67	0	0	2268	5848	0	75	5669	5999
P_mri-gridding	splitSort	2594	45	58	0	2594	2594	2594	0	16	2492	2588
R_hotspot	calculate_temp	1849	60	31	84	1808	1849	1849	0	0	456	1800
P_sad	mb_sad_calc	1584	120	14	0	0	1583	1583	0	0	37	356
R_backprop	bpnn_layerfwd_CU	4096	90	46	0	0	4096	4096	0	0	12	214
P_mri-gridding	binning_kernel	5188	45	116	1	1	1	1	5185	5188	5188	5188
R_hybridsort	mergeSortFirst	4098	90	46	1	1	1	1	3849	4072	4096	4098
R_kmeans	invert_mapping	3249	90	37	0	0	0	0	3153	3249	3249	3249
R_cfd	cu_time_step	1212	120	11	0	0	0	0	1149	1199	1211	1211
R_cfd	cu_initialize_vars	1212	120	11	0	0	0	0	1146	1200	1212	1212
R_srad_v1	prepare	450	45	10	1	1	1	1	354	445	449	450
R_srad_v1	srad	450	45	10	0	0	0	0	285	431	449	450
I_CP	cenergy	256	120	3	0	0	0	0	240	252	252	252
R_srad_v1	srad2	450	45	10	1	1	1	1	259	437	449	450
R_cfd	cu_comp_step_fact	1212	120	11	0	0	0	0	159	797	1185	1210
P_sad	larger_sad_calc_8	99	99	1	0	0	0	0	99	99	99	99
R_streamcluster	kernel_compute_cost	128	30	5	0	0	0	0	87	124	125	126
R_srad_v1	extract	450	45	10	1	1	1	1	68	394	450	450
R_srad_v2	srad_cu_1	16384	90	183	0	0	0	0	26	496	3209	8414
R_kmeans	kmeansPoint	3249	90	37	0	0	0	0	19	33	48	1743
R_cfd	cu_compute_flux	1212	45	27	0	0	8	252	6	66	654	1065
P_mri-q	ComputeQ_GPU	128	75	2	0	0	0	0	1	123	126	126
R_backprop	bpnn_adjust_wts_cu	4096	90	46	0	0	1	1	0	0	518	2864
R_srad_v2	srad_cu_2	16384	90	183	0	0	0	0	0	0	69	729
R_srad_v1	reduce	225	45	5	0	0	0	0	0	0	0	77
P_mri-gridding	scan_L1_kernel	41	41	1	41	41	41	41	0	0	0	0
R_pathfinder	dynproc_kernel	463	90	6	1	2	462	462	0	0	0	0
I_LPS	GPU_laplace3d	100	100	1	0	6	100	100	0	0	0	0
P_stencil	block2D_hyb_coarsen	64	64	1	4	4	4	4	0	0	0	0
P_cutcp	cu_cutoff_pot_lat	121	120	2	0	0	0	27	0	0	0	0
P_tpacf	gen_hists	201	45	5	0	0	0	11	0	0	0	0
I_AES	aesEncrypt128	257	90	3	0	0	0	0	0	0	0	4
I_NN	executeThirdLayer	2800	120	24	0	0	0	0	0	0	0	0
I_NN	executeFourthLayer	280	120	3	0	0	0	0	0	0	0	0
I_NN	executeSecondLayer	1400	120	12	0	0	0	0	0	0	0	0
I_STO	sha1_overlap	384	45	9	0	0	0	0	0	0	0	0
R_hybridsort	bucketsort	1024	120	9	0	0	0	0	0	0	0	0
R_hybridsort	bucketcount	1024	120	9	0	0	0	0	0	0	0	0
R_hybridsort	hist1024Kernel	64	60	2	0	0	0	0	0	0	0	0
R_nw	needle_cu_shared_1	64	15	5	0	0	0	0	0	0	0	0

kernel than a scheduler like GTO, which gives higher performance, at the expense of fairness across different warps.

Table 1 shows the warp divergence statistics[3]. The third column is the total number of TBs in the grid. The fourth column is the number of TBs that can be resident at a time (across all SMs) on the GPU. *Iters* [(column 5)] is column 3 divided by column 4, rounded to the next integer. Intuitively, if the total number of TBs is considered to be the total number of tasks and number of resident TBs as the number of workers, then each worker will do, on an average, *Iters* number of tasks.

The next 8 columns show (in 2 groups) the amount of divergence among warps of a TB measured in terms of number of dynamic instructions and number of cycles. For each warp, we measured the number of instructions executed by it and the number of cycles it took to complete kernel execution. We define instruction (cycle) divergence of a TB as the maximum difference between the dynamic instructions (cycles) executed by its constituent warps. A TB is said to have 10% or more instruction (cycle) divergence if its instruction (cycle) divergence is more than 10% of the minimum number of instructions (cycles) executed by one of its warps. For

[3] To reduce the table size, we have used P for Parboil, R for Rodinia and I for GPGPU-Sim benchmark suite names in the column BM, and we have shortened the kernel names.

Table 2: Cycles with finished warps

BM	Kernel	DWR	DWS
P_lbm	performStreamColl	0.58	0.50
R_hybridsort	mergepack	0.76	0.51
R_bfs	Kernel	0.75	0.65
R_bfs	Kernel2	0.51	0.33
R_hybridsort	mergeSortPass	0.23	0.20
I_BFS	Kernel	0.60	0.54
I_MUM	mummergpuKernel	0.40	0.33
I_NQU	solve_nqueen_cu	0.31	0.24
I_RAY	render	0.37	0.21
R_b+tree	findK	0.57	0.35
I_NN	executeFirstLayer	0.10	0.04
R_b+tree	findRangeK	0.32	0.2
P_mri-gridding	splitSort	0.06	0.05
R_hotspot	calculate_temp	0.17	0.06
P_sad	mb_sad_calc	0.06	0.03
R_backprop	bpnn_layerfwd_CU	0.09	0.02
P_mri-gridding	binning_kernel	0.55	0.53
R_hybridsort	mergeSortFirst	0.79	0.69
R_kmeans	invert_mapping	0.38	0.37
R_cfd	cu_time_step	0.63	0.55
R_cfd	cu_initialize_vars	0.55	0.48
R_srad_v1	prepare	0.50	0.41
R_srad_v1	srad	0.29	0.22
R_srad_v1	rad2	0.4	0.27
R_cfd	cu_comp_step_fact	0.76	0.58
P_sad	larger_sad_calc_8	0.11	0.11
R_streamcluster	kernel_compute_cost	0.05	0.04
R_srad_v1	extract	0.53	0.24
R_srad_v2	srad_cu_1	0.16	0.09
R_kmeans	kmeansPoint	0.09	0.03
R_cfd	cu_compute_flux	0.09	0.08
R_backprop	bpnn_adjust_wts_cu	0.18	0.07
R_srad_v2	srad_cu_2	0.04	0.03
R_srad_v1	reduce	0.04	0.01
R_pathfinder	dynproc_kernel	0.04	0.01
P_cutcp	cu_cutoff_pot_lat	0.44	0.09
I_AES	aesEncrypt128	0.1	0.06

example, kernel *render* has 108 TBs with instruction divergence of 10% or more, and 107 TBs with cycle divergence of 10% or more.

Based on instruction and cycle divergence, we have divided the benchmark kernels into 4 sets. The first set has kernels with both instruction and cycle divergence. Kernels in the second set have very little or no instruction divergence but significant cycle divergence. The third set contains kernels with instruction divergence but no cycle divergence. Kernels in the fourth set have very little or no divergence.

Next we bring out the effect of warp divergence on resource under utilization and lower warp residency. First we report the fraction of a TB's execution during which it experiences warp divergence. This, referred as *Divergent Warp Region*, is defined for a TB as

$$DWR(TB_i) = 1 - \frac{MinWarpCycles(TB_i)}{MaxWarpCycles(TB_i)}$$

where, MinWarpCycles and MaxWarpCylces correspond to the execution cycles of the earliest and the last finishing warp in TB_i. Column 3 in Table 2[4] reports *DWR* for the kernel

[4] Kernels with values less than 0.03 are not shown to reduce the table size.

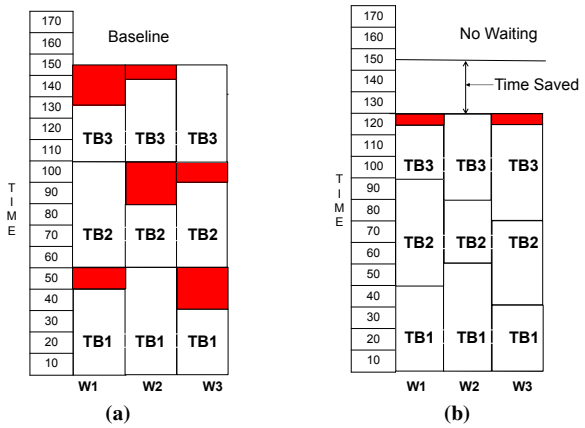

Figure 1: Effect of Warp Divergence on Performance

which is the average (arithmetic mean) across all TBs.

$$DWR(K) = \frac{1}{N-R}\sum_{i=1}^{N-R} DWR(TB_i)$$

where N is the total number of TBs in the execution configuration and R is the maximum number of TBs that be resident on the GPU at a time. We restrict the measurement of DWR to first $N - R$ finishing TBs, because after that there are no more TBs in the global TB scheduler waiting to be assigned to an SM.

Next we report the fraction of stall cycles in the divergent warp region. A stall cycle is one in which the warp scheduler can not schedule any warp. Column 4 in Table 2 reports stalls in divergent warp region (*DWS*) for the kernel which is defined as the ratio of number of stall cycles in the DWR region to the total number of stall cycles, averaged across all TBs. Intuitively, higher DWS points to the increase in stall cycles in DWR due to reduced number of ready warps. For example, for kernel *mergepack* 76% of total execution cycles are in the DWR region. Further, out of the total stall cycles, 51% cycles are in the DWR region, possibly due to lower warp residency. As Table 2 shows, kernels from the first two sets have larger values of *DWS* and these kernels experience warp divergence on a large number of TBs.

Next we illustrate in Figure 1, how *DWS* can be reduced if we can retain the warp residency during the divergent warp region. Assume a kernel with 3 TBs is invoked on a GPU with one SM. Also assume that only one TB can reside on the SM and the TB has 3 warps, W1, W2 and W3. Figure 1(a) shows the baseline behaviour of execution of TBs on the SM. Initially TB1 starts executing on the SM. Warps W1 and W3 finish before W2 but have to wait for W2 to finish. W1 waits for 10 time units and W3 waits for 20 time units and is shown in red colour. When warp W2 finishes execution at time 50, the resources allocated to TB1 are released and a new TB, TB2, starts executing. This time warp W2 waits for 20 time units and W3 for 10 time units. The second TB finishes execution at time 100. While one or

more warps are waiting, the warp scheduler may not have enough ready warps to hide long latencies. In this example, the last TB starts executing at cycle 101. The red portions from cycle 1 to cycle 100 show the cycles in which one or more warps are waiting for their siblings. This corresponds to column 3 in table 2, i.e., *DWR* is from cycles 31 to 50 and cycles 81 to 100.

Figure 1(b) shows the performance improvement that can be obtained if warps do not wait for their siblings. As soon as warp W3 of TB1 finishes at time 30, a new warp of TB2 starts executing, eliminating the waiting time. Similarly, as soon as warp W1 finishes at time 40, a new warp of TB2 starts executing. Since the grid has only 3 TBs, after warps W1 and W3 of TB3 finish, no new warps can start running in their place causing resources to be wasted which is shown by the red coloured blocks. As shown in Figure 1(b), the total runtime of the kernel reduces from 145 to 120 time units.

Our proposed solution tries to achieve this by using virtual TBs and virtual warps. So, for this example, one virtual TB with 3 virtual warps will be launched. Initially the virtual TB is assigned logical TB index 1 and the virtual warps are assigned logical warp indices 1 to 3. At cycle 30, virtual warp W3 finishes and it takes the next available TB and warp index, and is ready to execute the kernel code again starting from cycle 31. Similarly, virtual warp W1 takes the next available logical TB and warp index at cycle 41 and executes the kernel code again. This eliminates the waiting time (red portions) and also maintains residency which in turn helps to reduce the number of stall cycles. The proposed VTW approach enables code optimizations and also optimizations to warp schedulers as explained in the Section 4.

4. TB and Warp Virtualization

In this Section we explain in detail our proposed solution, *VTW*, to reduce the negative impact of warp divergence.

The execution configuration of a kernel specifies the number of TBs and number of threads in a TB. These are logical TBs and each logical TB has a unique index. This index can be obtained using the CUDA built-in variable *blockIdx*. Similarly, the unique thread index of a thread in its TB can be obtained using the CUDA built-in variable *threadIdx*.

The first part of VTW is a compiler transformation to use virtual indices instead of logical indices. The virtual indices are set to the appropriate logical indices at the beginning of the kernel code. When a warp finishes executing the kernel code, the virtual indices are assigned the next available logical indices and the kernel code is executed again. This process is repeated until all the user provided logical TB indices have been assigned. The second part contains two hardware changes, (1) to compute the next available logical indices, and (2) enhancements to the warp scheduler.

4.1 Code Transformations for Virtual TB and Warp

Now we discuss the transformations to convert a kernel to use virtual TB and thread indices, instead of the logical

```
__global__ void Fn(int*a, int*b, int*c) {
  int tIdx = threadIdx.x;
  int bIdx = blockIdx.x;                          1
  int idx = bIdx * blockDim.x + tIdx;
  c[idx] = a[idx] + b[idx];
}
```

```
__global__ void Fn(int*a, int*b, int*c) {
  int virtTbId = blockIdx.x;
  int logTbId = blockIdx.x;                       2
  int tIdxInBlk = threadIdx.x;
  while (logTbId < numLogicalTBs) {               5
    int thrdIdx_x_new = tIdxInBlk;
    int blkIdx_x_new = logTbId;                   3a
    int tIdx = thrdIdx_x_new;
    int bIdx = blockIdx_x_new;
    int idx = bIdx * blockDim.x + tIdx;           4
    c[idx] = a[idx] + b[idx];
    logTbId = _next_TbId;
    tIdxInBlk = _next_tIdxInBlk;                  3b
  }
}
```

Figure 2: Virtual TB and Warp Code Example

indices. Figure 2 shows the original and transformed code of a kernel. This example assumes both the grid and TB to be one dimensional. The upper part of the figure, marked as *1*, shows a sample code to add two arrays. Using *threadIdx.x* and *blockIdx.x*, it computes the flattened index *idx*.

The lower part of the figure shows the transformed kernel code. We call the warps and TBs of the transformed kernel, *virtual Warps* and *virtual TBs*. The total number of virtual TBs of the transformed kernel is computed as:

$$NumVirtualTBs = MIN(NewRes, NumLogicalTBs)$$

where, *NewRes* is the residency of the transformed kernel on the GPU and *NumLogicalTBs* is the total number of TBs of the original kernel.

In the transformed code, block *2* defines variables to track the virtual and logical indices. Code blocks *3a* and *3b* compute the next logical TB and thread index. Note that since the original kernel code uses only one dimension of threadIdx and blockIdx, code block *3a* shows only 2 new variables, viz., *thrdIdx_x_new* and *blkIdx_x_new*. These variables are used to set the appropriate logical thread and block indices. Instructions *_next_TbId* and *_next_tIdxInBlk* in code block *3b*, can either be implemented in hardware or software. We implemented them in hardware to reduce runtime overheads. The details are in Sections 4.2.1 and 4.2.2. One simple software implementation of *_next_TbId* is to increment *logTbId* by *NumVirtualTBs*, and of *_next_tIdxInBlk* is to return the same thread index.

The transformed code uses virtual thread and TB index variables instead of the CUDA built-in variables. Finally, the while loop marked as *5*, enables warps to acquire new indices and execute the kernel again using the new indices. We call this while loop, the *Wrapper While Loop* (WWL). Variable *logTbId* is a scalar variable and hence, if the original kernel code uses two or three dimensional grid, then to con-

vert the scalar *logTbId* value to the original two or three dimensional TB index, original grid dimensions i.e., *gridDim*, are needed. Similarly, to convert the scalar variable *tIdxInBlk* to the original multidimensional thread index, original TB dimensions, i.e., *blockDim*, are needed.

In the transformed code, as soon as a virtual warp finishes an iteration of WWL, it can acquire the next logical TB and thread index, and execute the kernel code again. This not only eliminates the waiting time of warps but also keeps the number of active warps constant. The only time a virtual warp waits for its siblings is when there are no more logical TB indices available and the warp exits the *WWL*.

Return statements in the original kernel code need to be replaced with an unconditional branch statement to jump to the beginning of block *3b*.

In order to prevent modifications to the shared memory by a virtual warp executing next iteration of the *WWL*, while its siblings executing the previous iteration may still be accessing the shared memory, certain modifications are needed which are discussed in Section 4.4.

4.2 ISA Extensions

Now we describe implementation of the new instructions to compute the logical TB index and thread index.

4.2.1 Computation of New Thread Block Index

When a virtual warp executes the instruction *_next_TbId*, it is assigned the next available logical TB index. To achieve this, the hardware keeps the mapping between the virtual TB index and logical TB index in each SM. It also maintains for the entire GPU, a count of the number of logical TB indices assigned so far, *nextLogTbId*, which starts at 0 and is incremented by 1 every time a new logical TB index is needed. This is similar to getting a new TB from the Thread Block Scheduler. The instruction *_next_TbId* is executed only by the first thread of each warp, other threads of the warp receive the value computed by the first thread. Along with the mapping between virtual and logical TB indices, the hardware also tracks the number of warps of a virtual TB that have been assigned the next logical TB index.

In order to simplify the hardware changes, we restrict the difference in iteration numbers of warps of a virtual TB to 1.

4.2.2 Computation of New Thread Index

After getting the next logical TB index, each thread gets the next logical thread index using *_next_tIdxInBlk*. We explored 3 different ways of getting the next logical thread index. In the first one, called *SameLane*, a warp maintains its relative position in the virtual TB and hence the thread indices of its constituent threads do not change. For example, warp 1 of a virtual TB, is always assigned warp index 1 of the logical TBs mapped to the virtual TB. In this case the instruction *_next_tIdxInBlk* simply returns the input thread index.

In the second case, called *SlowestLane*, warps of a virtual TB are sorted in the increasing order of the progress they have made. When a warp finishes an iteration of the *WWL* and if it is the first one to finish that iteration, all warps of the TB are sorted in increasing order of the progress they have made in the current iteration. The warp with the least progress is considered to be the slowest. A warp finishing its current iteration is assigned the next available warp id going from the slowest to the fastest warp. So the first warp of a virtual TB to start executing *i+1*th iteration is assigned the index of the slowest warp from *i*th iteration, the second warp is assigned the second slowest and so on. The intuition is that the warp with the least progress is likely to finish its current iteration last and hence it is beneficial to start it earlier in the next iteration.

The third approach of assigning thread indices, called *LowestToHighestLane*, assigns indices from the lowest index to the highest index. So, the first warp of a TB to finish an iteration is assigned warp index 0 for its next iteration, the second is assigned warp index 1, and so on.

4.3 Optimizations Enabled by VTW

4.3.1 Redundant Code Elimination

VTW enables loop independent code motion. Any code segment in the original kernel code that is independent of thread and TB indices can potentially be moved out of the wrapper while loop. This reduces redundant computation which otherwise happens in CUDA execution model [22], [33]. A side effect of the code transformation is the increase in number of live registers which is due to the extra variables used in index computation code and loop independent code motion.

4.3.2 Improvements to Warp Scheduler

So far, we have discussed how warps of a virtual TB can finish earlier and start executing the kernel code for a new logical TB. Here, we discuss how this information can be used in prioritising warps and TBs, and improve performance.

When all warps of a virtual TB have the same logical TB index, i.e., executing the same iteration of the *WWL*, the virtual TB is said to be in the *SameIter* state. Otherwise the virtual TB is said to be in *DiffIter* state. A virtual TB enters the *Barrier* state when one or more of its warps have reached a barrier and are waiting for their siblings.

The execution of a kernel can broadly be divided into two phases. The first phase is from the beginning of the kernel execution till the last logical TB index has been assigned, at which point the second phase starts. These phases are called the *Fast Phase*, and the *Slow Phase*, respectively [1].

Next we describe how we adapt the warp scheduling algorithm [1] for VTW. In the *Fast Phase*, warps from TBs in the *Barrier* state are given the highest priority, followed by warps at lower iteration count from TBs in the *DiffIter* state, followed by warps from TBs in the *SameIter* state and finally warps at higher iteration count from TBs in the *DiffIter* state. Algorithm 1 shows the steps in ordering warps. If there are multiple TBs in the *Barrier* state, then the TB with more warps waiting at a barrier is given more priority (*sort-*

Algorithm 1 Warp Scheduler

```
1: procedure orderWarps()
2: sortedWarps ← 0
3: barTbs ← sortBarTbs()
4: for all tb ∈ barTbs do
5:    barWarps ← sortBarTbWarps(tb)
6:    for all warp ∈ barWarps do
7:       add(sortedWarps, warp)
8:    end for
9: end for
10: diffIterTbs ← sortDiffIterTbs()
11: for all tb ∈ diffIterTbs do
12:    diffIterWarps ← sortDiffIterTbWarps(tb)
13:    for all warp ∈ diffIterWarps do
14:       add(sortedWarps, warp)
15:    end for
16: end for
17: sameIterTbs ← sortSameIterTbs()
18: for all tb ∈ sameIterTbs do
19:    sameIterWarps ← sortSameIterTbWarps(tb)
20:    for all warp ∈ sameIterWarps do
21:       add(sortedWarps, warp)
22:    end for
23: end for
24: diffIterTbs ← sortDiffIterTbs2()
25: for all tb ∈ diffIterTbs do
26:    diffIterWarps ← sortDiffIterTbWarps2(tb)
27:    for all warp ∈ diffIterWarps do
28:       add(sortedWarps, warp)
29:    end for
30: end for
31: end procedure
```

BarTbs). Ties are broken by giving higher priority to the TB with more progress. The progress of a TB is the sum of instructions executed by all its constituent warps. Within a TB in the *Barrier* state, warps with less progress are prioritised over warps with more progress (*sortBarTbWarps*). Together, these two prioritising schemes, reduce the waiting time of warps at barrier [1].

When a TB is in the *DiffIter* state, one or more of its warps have already started executing the kernel code for another logical TB index and hence are in the next iteration of the *WWL*. This means, the remaining sibling warps that are still executing the previous iteration of *WWL*, not only have to complete their current iteration but also have to complete the next iteration. For this reason, warps at lower iteration of TBs in *DiffIter* state are given higher priority. If there are multiple TBs in the *DiffIter* state, then TBs with fewer warps at lower iteration count are given higher priority (*sortDiffIterTbs*). Also within a TB in the *DiffIter* state, warps with less progress are prioritised over warps with more progress (*sortDiffIterTbWarps*) These two prioritising schemes, quickly move TBs from the *DiffIter* state to the *SameIter* state.

TBs in the *SameIter* state and their warps are assigned priorities directly proportional to their progress (*sortSameIterTbs*, *sortSameIterTbsWarps*). Prioritising warps and TBs this way, enables faster TBs to finish earlier. It also enables unequal progress among TBs and warps of TBs avoiding warps reaching long latency instructions close to each other in time. A good overlap of execution phases among the TBs and also within a TB can be achieved, which helps the warp scheduler in hiding long execution latencies.

Finally, for TBs in the *DiffIter* state, a TB with more warps at higher iteration count is given more priority (*sortDiffIterTbs2*).

The main reason behind giving the least priority to warps from TBs in the *DiffIter* state that are at higher iteration counts, is to schedule them only when there are no warps from TBs from the other 3 states. This reduces the interference caused by such warps to other warps. These are the warps that would have been waiting for their siblings in the baseline architecture but VTW makes them available for scheduling, improving the ability of warp schedulers to hide long latencies better.

The runtime of a kernel is decided by the TB finishing last. In the slow phase i.e., after the last logical TB index has been assigned to a virtual TB, warps and TBs are assigned priorities which are inversely proportional to their progress. Giving higher priority to warps with less progress helps to finish the entire TB faster. Also, giving higher priority to TBs with less progress enables them to finish earlier.

4.4 Shared Memory Access Restrictions

For kernels with shared memory accesses, if a virtual warp completes its current iteration and starts executing the kernel code for the next logical TB index, it cannot access shared memory until all its siblings executing the previous iteration, have gone past the last shared memory instruction in the kernel code. We achieve this by not allowing warps to schedule a shared memory instruction, until all of the sibling virtual warps complete the previous iteration. which ensures that all the shared memory accesses from the previous iterations are over. In order to use software versions of the instructions *_next_TbId* and *_next_tIdxInBlk*, before the first shared memory access in each iteration, a barrier synchronization statement is needed to make sure that virtual warps of a virtual TB from different iterations are not accessing the shared memory at the same time.

4.5 Hardware Overheads

We discuss the hardware overheads of VTW in the context of the NVIDIA Fermi architecture [9] which has 15 SMs, each SM allowing up to 8 TBs and 48 warps.

First, we discuss the extra hardware required for assigning new indices. VTW needs to maintain the virtual to logical TB index map on each SM, i.e., 1 integer per TB. Also, per virtual TB, a count of number of warps which are yet to be assigned the new logical index is needed, i.e., 1 byte

Table 3: GPGPU-Sim Configuration

Architecture	NVIDIA Fermi GTX480
Number of SMs	15
Max Number of TBs per SM	8
Max Number of Threads per Core	1536
Shared Memory per Core	48KB
L1-Cache per Core	16KB
L2-Cache	768KB
Max Number of Registers/Core	32768
Number of Warp Schedulers	2
DRAM Scheduler	FR-FCFS

per TB as a TB can have at most 32 warps. One integer variable for the GPU is needed to track the last assigned logical TB index. So, the extra storage needed to compute the next logical TB index is (8 * 4) + (8 * 1) + 4 = 44 bytes.

Out of the 3 approaches for getting the next logical thread index, *SameLane* approach does not need any extra storage. *LowestToHighestLane* approach needs a 1 byte variable to track the lowest unassigned warp index. *SlowestLane* sorts warps of a TB based on the progress of each warp and uses 48 1-byte registers to store the sorted warps. As the warp scheduler also uses the progress by each warp, we associate the extra storage cost of maintaining the progress of each warp with the warp scheduler.

The warp scheduler needs 1 byte per TB to track the number of warps of a TB in the next iteration. Also, It needs 48 registers to track the progress of each warp and 8 registers to track the progress of each TB. In addition, the warp scheduler needs 48 registers to track the current iteration indices of the 48 warps, 8 registers to track the lowest active iteration count of each TB. So, the extra storage is 1 * 8 + 4 * (48 + 8 + 48 + 8) = 456 bytes per SM.

So, the minimum extra storage needed for VTW is 500 bytes per SM for the *SameLane* approach and the maximum is 548 bytes per SM for the *SlowestLane* approach. This extra storage overhead is relatively small.

5. Experimental Evaluation

5.1 Experimental Methodology

We used the GPGPU-Sim [4] simulator version 3.2.2 to evaluate the proposed VTW technique. Table 3 shows the GPU configuration used in our simulation. We used benchmarks from GPGPU-SIM [4], Parboil [34] and Rodinia [5] benchmark suites. We compiled all the benchmarks using NVCC version 4.2 with default optimization level and used the PTX code for simulation.

5.2 Performance Analysis

Table 4 shows the performance of VTW. The performance numbers are normalized to the performance of original kernel with the GTO warp scheduling algorithm. Column *LRR* shows the performance of original kernel with the LRR warp scheduling algorithm. Columns *vGTO* and *vLRR* show the performance of transformed kernel, i.e. with virtual warps and virtual TBs, with the GTO and the LRR warp scheduling algorithms respectively. Column *VTW* shows performance of the transformed kernel with the progress aware warp scheduler described in Section 4.3.2. All the three, i.e. *vGTO*, *vLRR* and *VTW* use the *SameLane* mechanism for getting the next logical thread indices. IFinally, *vInstr* is the ratio of number of dynamic instructions executed by the transformed kernel to that of the original kernel.

In all our experiments, the original kernel is invoked with the user specified grid and the transformed kernel is invoked with as many TBs as allowed by the resource constraints. For example, for kernel *mergeSortPass* the user specified grid contains 695 TBs and for the transformed version of it, the grid contains 60 (virtual) TBs across all SMs.

VTW performs 6% better than GTO and 9% better than LRR. Even though *vGTO* and *vLRR* use the transformed kernel they do not show consistent performance improvements. Overall they slowdown by 2% compared to GTO. These performance numbers clearly show the effectiveness of our proposed optimization to the warp scheduling algorithm. All these 3 variants benefit from the reduction in the number of dynamic instructions executed. Overall, the transformed kernel executes 3% fewer instructions than the original kernel. It shows that the transformations enable compiler optimizations to eliminate redundant code. This data shows that all the components of our proposed technique viz., code transformations, compiler optimizations, ISA extension and warp scheduling optimizations, together improve performance.

The geometric mean performance improvement for each set shows *VTW* to be the best. On Set 1 which has kernels with both instruction and cycle divergence, *VTW* is able to extract an improvement of 2% over GTO. It is able to achieve as much as 37% improvement over GTO on kernel *calculate_temp*. VTW performs better on these kernels mainly due to improvements to the warp scheduling algorithm. On the kernels from Set 2, which exhibit only cycle divergence, *VTW* achieves as much as 76% improvement with an overall improvement of 5% over GTO. Again, the primary reason for *VTW* to perform better is the ability of its warp scheduling algorithm to utilize virtual warps better. In both these cases, even though on an average there is a reduction in the number of executed instructions, it does not always improve performance. For the 3rd set of kernels, number of executed instructions shows a direct impact on the performance. But even in this set of kernels, *VTW* benefits from its warp scheduling algorithm. Finally, for the fourth set of kernels, performance improvements are mainly due to the decrease in the number of executed instructions (due to the optimizations discussed in Section 4.3.1, and even *vGTO* and *vLRR* show considerable improvements.

Even though *vGTO* and *vLRR* use the same transformed kernel as *VTW*, they are slower than *VTW* by 8%. Only on 7 kernels *vGTO* is better than *VTW* with a maximum of 3% improvement. Also, *vLRR* is better than *VTW* on only 4 kernels. This clearly shows that the proposed warp

Table 4: Performance of VTW

Kernel	Performance				vInstr
	LRR	vGTO	vLRR	VTW	
performStreamColl	1.46	0.89	1.19	1.03	1.05
mergepack	1.00	0.91	0.81	1.01	1.12
Kernel	0.97	0.92	0.85	0.98	0.90
Kernel2	0.98	0.94	0.98	1.00	0.97
mergeSortPass	0.84	0.98	0.82	0.96	1.00
Kernel(I_BFS)	0.93	0.92	0.88	0.97	1.01
mummergpuKernel	0.98	0.99	0.95	1.01	1.01
solve_nqueen_cu	1.00	0.98	0.98	0.98	0.83
render	0.95	0.80	0.81	0.95	0.97
findK	0.93	1.16	1.09	1.14	1.12
executeFirstLayer	0.66	0.85	0.64	0.99	0.85
findRangeK	0.92	0.96	0.90	0.95	1.06
splitSort	0.91	1.05	0.91	1.03	1.00
calculate_temp	0.85	1.15	1.23	1.37	0.72
mb_sad_calc	0.95	0.97	0.94	0.97	1.04
bpnn_layerfwd_CU	0.97	0.97	0.95	1.01	1.05
GMEAN1	**0.95**	**0.96**	**0.92**	**1.02**	**0.98**
binning_kernel	0.94	0.95	0.90	0.96	1.05
mergeSortFirst	0.97	0.97	0.96	0.94	0.96
invert_mapping	0.93	1.01	0.91	0.99	0.99
cu_time_step	0.99	0.92	0.97	1.01	0.85
cu_initialize_vars	0.93	0.67	0.92	0.95	0.86
prepare	0.99	0.99	1.02	1.04	1.05
srad	0.99	0.96	1.00	1.06	1.04
cenergy	0.82	0.99	0.75	1.00	1.00
srad2	0.86	0.78	0.85	1.01	0.97
cu_comp_step_fact	1.00	0.94	1.05	1.02	0.94
larger_sad_calc_8	1.01	0.99	1.01	1.00	1.05
kernel_compute_cost	1.45	0.90	1.07	1.76	1.00
extract	0.91	0.89	1.02	1.03	1.09
srad_cu_1	0.89	0.90	0.94	1.07	0.91
kmeansPoint	0.98	0.91	0.94	0.99	1.00
cu_compute_flux	1.10	0.83	0.99	0.95	0.93
ComputeQ_GPU	0.93	1.10	0.93	1.13	0.90
bpnn_adjust_wts_cu	0.79	0.99	0.93	0.99	0.96
srad_cu_2	1.13	0.99	1.11	1.24	1.03
reduce	0.94	0.99	0.96	1.01	0.95
GMEAN2	**0.97**	**0.93**	**0.96**	**1.05**	**0.97**
scan_L1_kernel	0.97	1.03	1.02	1.06	0.93
dynproc_kernel	0.95	0.88	0.86	0.91	1.14
GPU_laplace3d	1.08	1.07	1.13	1.14	0.90
GMEAN3	**1.00**	**0.99**	**1.00**	**1.03**	**0.98**
block2D_hyb_coarsen	1.14	1.25	1.29	1.27	0.69
cu_cutoff_pot_lat	0.99	1.00	0.99	0.99	1.00
gen_hists	0.95	1.01	0.98	1.07	0.96
aesEncrypt128	0.89	1.00	0.92	1.03	1.01
executeThirdLayer	0.99	1.59	1.57	1.61	0.77
executeFourthLayer	1.01	1.47	1.49	1.51	0.78
executeSecondLayer	0.98	1.33	1.31	1.32	0.86
sha1_overlap	0.95	1.03	0.98	1.04	0.96
bucketsort	0.99	1.00	0.99	1.00	1.05
bucketcount	0.99	0.98	0.97	0.98	1.08
hist1024Kernel	1.00	0.98	0.98	0.98	1.15
needle_cu_shared_1	0.99	1.01	1.00	1.02	1.06
GMEAN4	**0.99**	**1.12**	**1.10**	**1.13**	**0.94**
GMEAN	**0.97**	**0.98**	**0.98**	**1.06**	**0.97**

scheduling algorithm of VTW utilizes the available ready warps in a better way to improve performance.

Table 5 shows performance of all three thread index assignment mechanisms. *VTW_SL* uses *SlowestLane* thread index assignment mechanism and *VTW_LH* uses *LowestToHighestLane*. Due to space constraints we show only the geometric mean improvements for each set and across all sets. From the table it is clear that all the three have similar performance. *SameLane* thread index assignment is the simplest

Table 5: Performance of thread index assignment mechanisms

	VTW	VTW_SL	VTW_LH
GMEAN1	1.02	1.02	1.03
GMEAN2	1.05	1.05	1.04
GMEAN3	1.03	1.03	1.03
GMEAN4	1.13	1.14	1.13
GMEAN	1.06	1.06	1.06

Table 6: Fraction of Stall Cycles

	LRR	vGTO	vLRR	VTW	VTW_SL	VTW_LH
GMEAN1	1.12	1.05	1.13	0.96	0.95	0.95
GMEAN2	1.04	1.11	1.02	0.95	0.95	0.94
GMEAN3	0.96	1.00	1.01	0.90	0.90	0.90
GMEAN4	1.00	0.88	0.89	0.87	0.87	0.87
GMEAN	1.05	1.02	1.02	0.93	0.93	0.93

to implement with no extra hardware needed to compute the new thread index.

Table 6 shows reduction in stall cycles with the proposed warp scheduling algorithm. The data is normalized to the stall cycles of the original kernel with the GTO warp scheduling algorithm (lower numbers means fewer stalls and hence are better). Due to space constraints, we show only the geometric mean reduction in stall cycles for each set and across all sets. The number of stall cycles used for computing the fractions is the average across all SMs on the GPU. As can be seen from the geometric mean numbers, VTW shows reduction in the number of stall cycles compared to both the GTO and LRR warp scheduling algorithms on the original as well as the transformed kernel. Overall, there is 7% reduction in stalls with VTW over GTO. This reduction results in commensurable performance improvement.

Performance of *VTW* is less than GTO by more than 5% in only two kernels, and the maximum slowdown is 9% in kernel *dynproc_kernel*. But for that kernel, the average number of stall cycles with *VTW* is less than GTO by 6%. This can happen because the total runtime of a kernel is the time to execute all TBs and sometimes there are a small number of long running TBs. For example, kernel *solve_nqueen_cu_kernel* is invoked with 256 TBs out of which 255 TBs finish in first 6473 cycles but the last TB finishes at cycle 35115. As explained before, *VTW* prioritises slower warps and TBs in the *slowPhase*. If one of the TBs with less priority happens to be a long running TB then its runtime may increase, causing the total runtime of the kernel to increase. This was observed in kernel *render*.

5.3 Analysis of Compiler Transformations

Table 7 shows the effect of our proposed compiler transformations on the number of registers per thread and residency of the GPU. The table shows only those kernels where the change in number of registers impacted the residency. The second and the third columns show the number of registers per thread for the original and transformed kernel, re-

Table 7: Effect on Number of Registers and Residency

Kernel	Regs	vRegs	Res	vRes
mergeSortPass	28	30	75	60
findRangeK	19	22	90	75
cu_time_step	18	22	120	105
srad2	18	22	45	30
cu_comp_step_fact	19	23	120	105
srad_cu_1	19	23	90	75
kmeansPoint	17	22	90	75
ComputeQ_GPU	21	20	75	90
bpnn_adjust_wts_cu	18	24	90	75
srad_cu_2	17	22	90	75

spectively. The fourth and the fifth columns show the maximum number of resident TBs per GPU for the original and transformed kernel, respectively. Reasons for increase in the number of registers are the extra variables introduced by the transformation and loop independent code motion.

VTW enables control over the resources used by controlling the number of virtual TBs launched. This can help to improve throughput in the case of concurrent kernels. We leave exploration of this to future work.

6. Related Work

One of the earliest works on warp divergence is by Xiang et al. [36]. They proposed a hardware solution to dispatch threads to an SM at the warp granularity. This enables them to have a partial TB executing on the SM. As soon as a warp finishes execution, its resources are released and a new warp from either a partially allocated TB, if there is one, or a new TB is allowed to start execution. One limitation of their solution is that the shared memory is allocated and deallocated at the TB level which means a new TB can be partially allocated only if its shared memory requirement is satisfied. In comparison with this, VTW allows multiple partial TBs i.e. there can be multiple virtual TBs with a subset of warps executing the kernel code for a different logical TB index than their siblings. Also, VTW reuses the shared memory assigned to a virtual TB and hence, as soon as a virtual warp finishes, it can acquire new logical TB index and start executing. Further, VTW optimizes the warp scheduler to intelligently use the faster warps.

CAWS [20] proposes techniques to prioritise critical warps to reduce the execution time disparity among warps within the same TB. Lee et al. [21] proposed a predictor to identify critical warps to prioritise them, and a cache reuse predictor that retains latency-critical blocks in the L1 data cache. VTW identifies critical warps using their progress. Also unlike, CAWS and CAWA, it eliminates waiting time of warps for their siblings.

Gupta et al. [12] characterized and analyzed Persistent Threads and compared them with the traditional GPU programming style. Stratton et al. [32], [33] discuss the idea of using a single CPU thread to execute multiple CUDA threads by inserting an enclosing iterative loop. VTW uses an enclosing iterative loop to iterate over TBs instead of threads, and the new TB indices are dynamically chosen. Elastic Kernel [26] uses the idea of iterative loop to generate elastic kernels which can be run with any number of TBs as well as any TB dimension. VTW does not change the size of TB.

A lot of research work focuses on warp scheduling. Narasiman et al. [24] proposed a two-level warp scheduler to hide long execution latencies in a better way. Gebhart et al. [11] proposed a two level warp scheduler focusing on energy efficiency. PRO [1] discusses a progress aware warp scheduling algorithm to reduce negative impact of long latency instructions and warp divergence. Compared to these, VTW proposes compile-time (code transformations) and run-time techniques to improve performance.

A lot of research focuses on improving performance by improving cache and memory performance [13], [14], [15], [29], [30], [31]. Lee et al. [19] proposed a warp scheduler, iPAWS, that dynamically adapts between a greedy and a round-robin policy, based on the instruction issue pattern. Awatramani et al. [3] identify phases of execution and their lengths at compile time. The warp scheduling policy chooses a warp with the shortest length for its next phase. Kim et al. [18] proposed a pre-execution mode in which warps stalled on long-latency operations can pre-execute successive independent instructions.

Yoon et al. [37] proposed an architecture in which as many TBs are assigned to an SM as allowed by the register and shared memory limits, ignoring the limits imposed by number of threads and TBs. Wu et al. [35] discuss SM-centric transformations for program-level spatial scheduling on the GPU (which SMs) and precise control of job locality on SMs (which TBs on which SM). Our proposed transformations are similar but with different goals. Chen et al. [6] proposed a compiler transformation to replace subkernel launches with code to be executed by the parent threads.

Various techniques to handle thread divergence have been proposed in [2], [8], [10], [17], [23], [27], [28].

7. Conclusion

We proposed VTW, a simple and elegant technique to improve performance of kernels that experience warp divergence. VTW creates virtual warps and virtual TBs using persistent threads and enables them to execute the kernel code for multiple logical TBs. We also proposed improvements to the warp scheduling algorithm to make them aware of the progress of virtual warps and TBs, and use this information to schedule warps effectively. Our proposed technique achieves 6% improvement over GTO and 9% improvement over LRR warp scheduling algorithms.

Acknowledgements

We thank the anonymous reviewers for their suggestions and comments. We also thank members of the Lab for HPC for discussions and feedback on improving the paper. The first author acknowledges the funding received from Google India Private Limited.

References

[1] J. Anantpur and R. Govindarajan. *PRO: Progress Aware GPU Warp Scheduling Algorithm*. IPDPS-2015

[2] J. Anantpur and R. Govindarajan. *Taming Control Divergence in GPUs through Control Flow Linearization*. CC-2014

[3] M. Awatramani , X. Zhu, J. Zambreno and D. Rover. *Phase Aware Warp Scheduling: Mitigating Effects of Phase Behavior in GPGPU Applications*, PACT-2015.

[4] A. Bakhoda, G. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt . *Analyzing cuda workloads using a detailed gpu simulator*. ISPASS-2009.

[5] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. H. Lee, and K. Skadron. *Rodinia: A benchmark suite for heterogeneous computing*. IISWC-2009.

[6] G. Chen and X.Shen. *Free Launch: Optimizing GPU Dynamic Kernel Launches through Thread Reuse*, MICRO-2015.

[7] CUDA. *CUDA C Programming Guide*.

[8] G. Diamos, B. Ashbaugh S. Maiyuran A. Kerr H. Wu and S. Yalamanchili. *SIMD Re-Convergence At Thread Frontiers*. MICRO-2011

[9] Fermi. http://www.nvidia.in/content/PDF/fermi_white_papers /NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

[10] W. Fung and T. Aamodt. *Thread Block Compaction for Efficient SIMT Control Flow*. HPCA-2011

[11] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, K. Skadron. *Energy-efficient Mechanisms for Managing Thread Context in Throughput Processors*. ISCA-2011

[12] K. Gupta, J. A. Stuart and J. D. Owens. *A Study of Persistent Threads Style GPU Programming for GPGPU Workloads*, InPar-2012

[13] A. Jog, O. Kayiran, N. C. Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, C. R. Das. *OWL: cooperative thread array aware scheduling techniques for improving GPGPU performance*. ASPLOS-2013.

[14] A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das. *Orchestrated scheduling and prefetching for gpgpus*. ISCA-2013.

[15] O. Kayiran, A. Jog, M. T. Kandemir and C. R. Das. *Neither More Nor Less: Optimizing Thread-level Parallelism for GPGPUs*. PACT-2013.

[16] Kepler. http://www.nvidia.in/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

[17] F. Khorasani, R. Gupta and L. N. Bhuyan. *Efficient Warp Execution in Presence of Divergence with Collaborative Context Collection*, MICRO-2015.

[18] K. Kim, S. Lee, M. K. Yoon, G. Koo, W. W. Ro, M. Annavaram. *Warped-Preexecution: A GPU Pre-execution Approach for Improving Latency Hiding*. HPCA-2016

[19] M. Lee, G, Kim, J. Kim, W. Seo, Y. Cho, S. Ryu. *iPAWS: Instruction-Issue Pattern-based Adaptive Warp Scheduling for GPGPUs*. HPCA-2016

[20] S. Lee and C. Wu. *CAWS: Criticality-Aware Warp Scheduling for GPGPU Workloads*. PACT-2014

[21] S. Lee, A. Arunkumar and C. Wu. *CAWA: coordinated warp scheduling and cache prioritization for critical warp acceleration of GPGPU workloads*. ISCA-2015

[22] Y. Lee, R. Krashinsky, V. Grover, S. W. Keckler and K. Asanovic. *Convergence and Scalarization for Data-Parallel Architectures*, CGO-2013.

[23] J. Meng, D. Tarjan and K. Skadron. *Dynamic Warp Subdivision for Integrated Branch and Memory Divergence Tolerance*. ISCA-2010

[24] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, Y. N. Patt. *Improving GPU Performance via Large Warps and Two-Level Warp Scheduling*. MICRO-2011.

[25] OpenCL. *www.khronos.org/opencl*.

[26] S. Pai, M. J. Thazhuthaveetil and R. Govindarajan. *Improving GPGPU Concurrency with Elastic Kernels*, ASPLOS-2013.

[27] M. Rhu and M. Erez. *CAPRI: Prediction of Compaction-Adequacy for Handling Control-Divergence in GPGPU Architectures*. ISCA-2012.

[28] M. Rhu and M. Erez. *The dual-path execution model for efficient gpu control flow*. HPCA-2013.

[29] T. G. Rogers, M. OConnor, and T. M. Aamodt. *Cache-conscious wavefront scheduling*. MICRO-2012.

[30] T. G. Rogers, M. OConnor, and T. M. Aamodt. *Divergence-Aware Warp Scheduling*, MICRO-2013

[31] A. Sethia, D. A. Jamshidi, S. Mahlke. *Mascar: Speeding up GPU Warps by Reducing Memory Pitstops*. HPCA-2015

[32] J. A. Stratton, S. S. Stone, and W. M. Hwu. *MCUDA: An efficient implementation of CUDA kernels for multi-core CPUs*, LCPC-2008.

[33] J. A. Stratton, V. Grover, J. Marathe, B. Aarts, M. Murphy, Z. Hu and W. W. Hwu. *Efficient compilation of fine-grained SPMD-threaded programs for multicore CPUs*, CGO-2010.

[34] J. A. Stratton, C. Rodrigues, I. Sung, N. Obeid, L. Chang, N. Anssari, G. D. Liu, W. W. Hwu. *Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing*. UIUC, Tech. Rep. IMPACT-12-01, March 2012

[35] W. Wu, G. Chen, D. Li, X. Shen and J. Vetter. *Enabling and Exploiting Flexible Task Assignment on GPU through SM-Centric Program Transformations*, ICS-2015.

[36] P. Xiang, Y. Yang, H. Zhou. *Warp-Level Divergence in GPUs: Characterization, Impact, and Mitigation*. HPCA-2014.

[37] M. K. Yoon, K. Kim, S. Lee, W. W. Ro, and M. Annavaram. *Virtual Thread: Maximizing Thread-Level Parallelism beyond GPU Scheduling Limit*, ISCA-2016