Warp-Level Divergence in GPUs: Characterization, Impact, and Mitigation

Ping Xiang, Yi Yang*, Huiyang Zhou

Dept. of Electrical and Computer Engineering
North Carolina State University
Raleigh, NC, USA
{pxiang, hzhou}@ncsu.edu

**Dept. of Computing Systems Architecture*
NEC Laboratories America
Princeton, NJ, USA
yyang@nec-labs.com

Abstract

High throughput architectures rely on high thread-level parallelism (TLP) to hide execution latencies. In state-of-art graphics processing units (GPUs), threads are organized in a grid of thread blocks (TBs) and each TB contains tens to hundreds of threads. With a TB-level resource management scheme, all the resource required by a TB is allocated/released when it is dispatched to / finished in a streaming multiprocessor (SM). In this paper, we highlight that such TB-level resource management can severely affect the TLP that may be achieved in the hardware. First, different warps in a TB may finish at different times, which we refer to as 'warp-level divergence'. Due to TB-level resource management, the resources allocated to early finished warps are essentially wasted as they need to wait for the longest running warp in the same TB to finish. Second, TB-level management can lead to resource fragmentation. For example, the maximum number of threads to run on an SM in an NVIDIA GTX 480 GPU is 1536. For an application with a TB containing 1024 threads, only 1 TB can run on the SM even though it has sufficient resource for a few hundreds more threads. To overcome these inefficiencies, we propose to allocate and release resources at the warp level. Warps are dispatched to an SM as long as it has sufficient resource for a warp rather than a TB. Furthermore, whenever a warp is completed, its resource is released and can accommodate a new warp. This way, we effectively increase the number of active warps without actually increasing the size of critical resources. We present our lightweight architectural support for our proposed warp-level resource management. The experimental results show that our approach achieves up to 76.0% and an average of 16.0% performance gains and up to 21.7% and an average of 6.7% energy savings at minor hardware overhead.

1. Introduction

Graphics processing units (GPU) or single-instruction multiple- thread (SIMT) architectures mainly rely on thread-level parallelism (TLP) to achieve high computational throughput and high energy efficiency. In state-of-art GPUs, threads are organized in a two-level hierarchy: multiple threads are grouped into a thread block (TB), and many TBs constitute a thread grid. Lightweight synchronization and data communication through shared memory are supported among the threads in the same TB. In research literature [3][13][15][24][25] and microarchitectural simulators [1], it is assumed that GPUs manage their resources at the TB granularity for general purpose GPU computing: a TB is dispatched to a streaming multiprocessor (SM) if it can allocate all the required resources of a TB, including the register file, the shared memory, the warp scheduler entries, etc. All the allocated resources for a TB are released when all the threads or warps in the TB complete execution. The limitation of TB-level resource management is first recognized in [23] and a solution has been proposed. In Section 2, we perform microbenchmarking on state-of-art NVIDIA GPUs and the results show that they are equipped with finer-grain resource management than the TB level, utilizing the approach from [23].

In this paper, we highlight that TB-level resource management can severely limit the TLP, i.e., the number of actively running threads or warps, which can be supported by the hardware. There are two fundamental reasons. First, as the threads in a TB are formed into multiple warps, in the absence of synchronizations each warp is executed independently to each other and different warps in a TB can finish at quite different times. In this paper, we refer to this divergent behavior as 'warp-level divergence'. Due to TB-level resource management, the resources allocated to early completed warps are essentially wasted as they need to wait for the longest running warp in the same TB to complete. Such resource underutilization can be viewed as 'temporal resource underutilization' as the resources are allocated but not reclaimed and reused promptly. We characterize warp-level divergence and temporal resource underutilization on a variety of applications and show that there exists a high potential if we can reclaim the resources of these early completed warps.

Second, TB-level resource management leads to resource fragmentation. Given a fixed size of a

physical resource, e.g., a 128kB register file (or 32k 32-bit scalar registers) in an SM, if a TB needs 9k registers (or 36kB), the register file will limit the maximal number of TBs to be dispatched to the SM to be 3. As the remaining part of the register file (128kB –3*36kB = 20kB or 5k registers) is not sufficient to support another TB, it will be always un-utilized. Such resource underutilization can be viewed as 'spatial resource underutilization' as the resource is never allocated. Besides the register files, the spatial resource underutilization will also happen for shared memory as well as the warp scheduler, which is designed for the maximal number of threads/warps on an SM.

In this paper, we propose a novel warp-level resource management scheme, referred to as 'WarpMan', to overcome the limitations of TB-level resource management. Our proposed approach enables a warp to be dispatched and executed in an SM if it has enough resources for a warp. Moreover, a warp can release its allocated resources as soon as it finishes execution. This way, we effectively increase the number of actively running warps without actually increasing the size of any critical physical resource including the register file, the shared memory, the maximal number of threads, and the maximal number of TBs on an SM. In other words, we simply make more efficient use of existing resources. We present our proposed architectural support for WarpMan and show that it only incurs minor changes to the existing hardware for TB-level resource management. We discuss the relationship between our approach and [23] in Section 2 and Section 7.

We observe that in some cases, increasing the number of actively running warps may cause contentions at caches as well as off-chip row buffers, thereby hurting performance. Both an SM-dueling technique and a heuristic-based approach are proposed to address this issue.

We model our proposed approach in GPGPUSim V3.2.0. Our experimental results based on the configuration of NVIDIA GTX480 GPUs show that our approach can greatly improve the performance for a variety of benchmarks, up to 76.0% and 16.0% on average, and reduce energy consumption by up to 21.7% and 6.7% on average.

Overall, our work makes the following contributions. (1) We highlight the limitations of TB-level resource management; (2) we characterize warp-level divergence and reveal the fundamental reasons for such divergent behavior; (3) we propose our warp-level resource management scheme and show that it can be implemented with minor hardware changes; (4) we show that our proposed solution is highly effective and achieves significant performance improvements and energy savings.

The rest of the paper is organized as follows. Section 2 gives a brief background of GPU computing and presents our experiments on actual GPUs to reverse engineer the resource management used in state-of-art GPUs. Section 3 dissects the limitations of TB-level resource management. Section 4 describes our architectural designs for WarpMan. Our experimental methodology and results are addressed in Section 5 and Section 6, respectively. Related works are discussed in Section 7. Section 8 concludes the paper.

2. Background

State-of-art GPUs exploits many-core architecture. A GPU has multiple streaming multiprocessors (SMs)/compute units (CUs) using the NVIDIA/AMD terminology. Each SM/CU includes one or more arrays of streaming processors (SPs)/thread processors (TPs). Each array of SPs/TPs follow the same instruction multiple data (SIMD) paradigm to execute a batch of threads/work items at a time. Such a batch of threads/work items is referred to as a warp/wavefront in NVIDIA/AMD architecture. Threads/work items of a GPU program follow the same program multiple data (SPMD) programming model. A typical general-purpose GPU program, also called a GPU kernel, is expected to have a large number, thousands or millions, of threads/work items. In order to manage such a high number of threads/work items, threads/work items are first grouped into thread blocks (TBs)/workgroups. Many TBs/workgroups are then organized into a grid. Threads/work items in the same TB/workgroup are executed on the same SM so that they can communicate with each other using the on-chip shared memory in the SM. As a result, it is convenient to use a TB/workgroup as the basic unit to be dispatched to a SM/CU.

With TB-level resource management, a GPU kernel dispatcher maintains the GPU kernel information, which includes the overall number of TBs, the next TB identifier (id) to be dispatched, and other related information. When a GPU kernel is launched, or a TB is completed in a SM, the kernel dispatcher will check whether the required resources of a TB can be satisfied by one of the SMs. These resources include the shared memory, the number of registers, and the number of warps in a TB. If all these resources requirements are satisfied by an SM, one TB will be dispatched onto it. Then, the kernel dispatcher will update the next TB id to be dispatched. After a TB is dispatched onto the SM, all the warps of the TB are eligible for execution. All the allocated resources are not released until all the warps in a TB have finished execution.

To analyze the resource management in state-of-art GPUs, we performed the experiments on both GTX480

(Fermi) and GTX680 (Kepler) GPUs. The kernel code used in this experiment is shown in Figure 1. The grid dimension is set as 1000 (i.e., 1000 TBs) and the TB size is set as 256 (i.e., 8 warps). The function 'bloatOccupancy' at line 1 ensures that each thread uses 32 registers. On a GTX 480 GPU, each SM has a 128KB register file (i.e., 32k registers) while each TB requires 8k (=32x256) registers. Therefore, the maximum occupancy for this kernel would be 4 TBs per SM if a TB-level resource management scheme is used.

```
1. __global__ void TB_resource_kernel(…, bool call = false){
2.    if(call) bloatOccupancy(start, size);
      ...
3.   clock_t start_clock = clock();
4.   if(tid < 32){ //tid is the threadid within a TB
5      clock_offset = 0;
6.      while( clock_offset < clock_count ) {
7.          clock_offset = clock() - start_clock;
8.      }
9.   }
10.   clock_t end_clock = clock();
11.   d_o[index] =  start_clock; //index is the global thread id
12.   d_e[index] =  end_clock;
13.}
```

Figure 1. A code example to confirm the TB-level resource management on GPUs

The statement at line 4, 'if(tid < 32)' where 'tid' is the thread idx within a TB, forces the first 32 threads (or the first warp) in a TB to delay for an amount of time specified with the variable 'clock_count'. The starting and ending times of all the threads are stored in global memory arrays. In this experiment, the variable 'clock_count' is set as (30*deviceProp.clockRate) such that the delay is always 30ms across different GPUs.

We checked the starting and finishing times of all the TBs for this kernel. The results shown in Figure 2 prove that at most 6 TBs (2 more than the maximum occupancy determined by the physical register file) can run concurrently on each SM. This indicates that the current GPUs may have some form of warp-level resource management. One likely reason is due to a patented register management scheme [23], in which a virtual register table is used to allow virtual registers to share physical registers. This register file management scheme makes the physical register file to appear larger and the virtual register table size instead of the physical register file size determines how many TBs can be accommodated on an SM. The difference from our proposed mechanism is that we enable more than 6 concurrent TBs on each SM. In each TB, there is one long-running warp and the remaining 7 warps finish quickly. Using our proposed approach, the quickly finished 7 warps will release their resource and new warps can be dispatched. After a while, there will be 8 concurrent TBs (1 long-running warp in each TB) running on an SM, where 8 is the upper limit of TBs on each SM on GTX480 GPUs. If ignoring this limit, 48 TBs (or 48 long-running warps) can run concurrently on each SM, where the maximum number of threads (1536 = 48x32) limits how many warps can run concurrently on each SM.

With the same kernel code, if the TB size is changed from 256 to 1024, the same experiment shows that there is at most 1 TB running at any time on an SM on GTX480 GPUs. With our proposed approach, many more TBs can run concurrently on each SM for the same reason discussed above. The experiments on GTX680 GPUs show very similar results.

```
> Using CUDA device [0]: GeForce GTX 480
> Detected Compute SM 3.0 hardware with 8 multi-processors.
…
CTA 250
 Warp 0: start 80, end 81
CTA 269
 Warp 0: start 80, end 81
CTA 272
 Warp 0: start 80, end 81
CTA 283
 Warp 0: start 80, end 81
CTA 322
 Warp 0: start 80, end 81
CTA 329
 Warp 0: start 80, end 81
…
```

Figure 2. Starting and ending times for different TBs/warps on GTX 480 for the kernel shown in Figure 1.

3. Limitations of Thread-block Level Resource Management

3.1. Spatial Resource Underutilization Due To Thread-Block Level Resource Allocation

With TB-level resource management, during TB dispatch, an SM tries to allocate the necessary resources at the TB granularity. Therefore, an SM cannot launch a TB unless its available resources can accommodate the whole TB. In other words, no partial TB can be dispatched to an SM. This can result in resource fragmentation or un-utilized resources. We refer to this type of resource underutilization as spatial resource underutilization. On an SM, the critical resources, which can be spatially underutilized due to the TB-level resource allocation, include the register file, the shared memory, and the warp scheduler, whose size determines the maximal number of threads/warps that can run on an SM. Next, we use the register file resource as an example and analyze a set of GPU applications (see Section 5 for methodology). Among these applications, six benchmarks, RS, HS, RAY, MM, NN and CT, are sensitive to the register file size as the number of TBs that can be dispatched to an SM, i.e., TLP, is determined by their register usage. Another four benchmarks, BT, SN, SR and PF, are not register file sensitive and the maximal number of TBs that can be dispatched is limited instead by the number

of threads in a TB (i.e., the warp scheduler size limits the number of TBs to be dispatched to an SM). For the remaining benchmarks, ST, HG, and MC, the maximal number of TBs to be dispatched to an SM is limited by both its register usage and the number of threads in a TB. Among the benchmarks, seven of them use the shared memory but the shared memory size (48KB per SM) is not the limiting factor for TLP.

Figure 3 shows the ratio of the number of registers allocated for the dispatched TBs on an SM over the total number of registers on the SM for different applications. In this experiment, we use the configuration of NVIDIA GTX 480 GPUs: the register file size is 128kB; the maximal number of threads that can run on an SM is 1536; and the maximal number of TBs to be dispatched to an SM is 8. For each benchmark, we include the number of TBs that can be dispatched to an SM besides the benchmark name. Different colors are used in Figure 3 to represent the registers allocated for different TBs.

From Figure 3, we can see that many applications do not fully utilize the precious register resource. For the benchmarks in our study, an average, using the geometric mean (GM), of 29.6% of registers is not used at all during program execution. This observation is also made in [15] and used for static energy savings. Our new and interesting observation is that even for the register file sensitive applications, i.e., the applications that would benefit from large register files, the register file remains underutilized and the fundamental reason is due to TB-level resource management. For the benchmark, RS, each of its TBs needs 11776 registers (or 35.9% of the register file). Therefore, each SM can support two TBs and the remaining 28.1% register file (or 9216 registers) is un-utilized. The other four benchmarks, MM, SN, RAY and NN, also show similar behavior to RS, and 6.3%, 20.3%, 25.0% and 6.3% of the registers are never used in each SM, respectively. For the remaining register file sensitive application, CT, although its TLP is limited by its register usage, the un-utilized register number is very small (512 registers or 1.6% of the total register file). Overall, due to such coarse grained register allocation, on average, 13.6% of the total registers are unutilized for these six register file sensitive benchmarks (labeled as ‘GM_reg’ in Figure 3). For MC, HG and ST, as their TLP is limited by both the registers and the warp scheduler size (i.e., the maximal number of threads), a larger register file alone will not enable more TBs to be dispatched to an SM. For the benchmarks that are not register file sensitive, including BT, SN, SR, and PF, although a significant part of register file is un-utilized, the register file is not the resource bottleneck for higher TLP anyway. In other words, enlarging the register file will result in more idle registers.

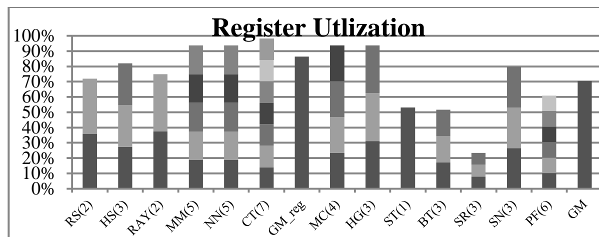

Figure 3. The ratio of allocated number of registers over the total number of registers. The number besides the benchmark name is the number of TBs that can run on an SM, simultaneously.

3.2 Temporal Resource Underutilization Due To Thread-Block Level Resource Release and Reallocation

TB-level resource management requires that all the allocated resources for a TB are released only when all the threads/warps in the same TB finish the execution. At run-time, although the warps in the same TB start at the same time, they may take different amounts of execution times to complete. We refer to this behavior as warp-level divergence as opposed to the well-known branch divergence or thread-level divergence. Next, we characterize warp-level divergence and reveal that it can be caused by: (1) input-dependent workload imbalance; (2) code-dependent workload imbalance; (3) execution divergence such as memory divergence; and (4) warp-scheduling policies.

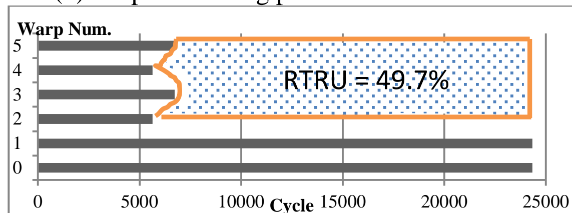

Figure 4. The lifetimes/execution times of different warps in TB0 of the benchmark RAY. The ratio of temporal resource underutilization (RTRU) is 49.7%.

To quantify the impact of warp-level divergence, we propose the following metric. First, for a TB with N warps, we collect the execution time for each of its warps, T_i for warp $_i$. The longest execution time among the warps in the TB is denoted as *maxT*. Then, we accumulate the wasted time for all the warps in the TB as $\sum(maxT\text{-}T_i)$ and then normalize the sum to $(N*maxT)$. The resulting ratio, $\sum(maxT\text{-}T_i) / (N*maxT)$, essentially represents how much resource is wasted due to warp-level divergence. Therefore, we refer to it as the ratio of temporal resource underutilization (RTRU). Using the benchmark RAY as an example, the lifetimes or execution times of all the warps in TB0, are shown in Figure 4. The RTRU for this TB, $\sum(maxT\text{-}T_i) / (N*maxT)$, is the shaded area in Figure 4

(49.7%). When an application has many TBs, we use the geometric mean to compute the average RTRU. For RAY, the average RTRU among its 512 TBs is 39.2%.

3.2.1. Input-dependent workload imbalance. In the SIMT programming model, different threads process different data elements based on their thread ids. If the program performs different operations dependent upon the input data values, different threads or warps will perform different operations and both thread-level divergence and warp-level divergence can happen. We show such an example in Figure 5 and the code is extracted from the benchmark RAY.

```
__global__  render(Plane){
  …//Generate Ray based on thread_id and block_id;
  int n(0), nRec(5);
  for( int i(0); i < nRec && n == i; i++ ) {
       …
       t  = intersectionPlan(Ray, Plane)
       …
       if( t > 0.0f && t < 10000.0f ) {
          n++;
          …
       }  }  }
```

Figure 5. A code segment from the benchmark RAY.

From Figure 5, we can see that depending on the value of the input data (i.e., '*Plane*'), which are used in the function '*intersectionPlan*' to compute intersection between plane and rays, different threads/warps may choose different control paths at the '*if*' statement inside the loop, which then results in the '*for*' loop being iterated for different numbers of times for different warps. Therefore, different warps may have different dynamic instruction counts depending on the input data. For this benchmark, the input, i.e., the objects and scene, is the one provided in GPGPUsim [1]. To analyze such input-dependent workload imbalance, in Figure 6, we report the histogram of the dynamic instruction counts for different warps in the same TB.

From Figure 6, we can see that 52.5% of the warps execute less than 500 instructions while 29.6% of warps execute 1000 to 1500 instructions. About 1.9% of the warps actually execute 3500 to 4000 instructions. Such variances in the dynamic instruction count account for different warp lifetimes. For TB0, the lifetimes for its six warps are shown in Figure 4. As we can see from Figure 4, for the first 6000 cycles, all the warps in the TB are busy. At cycle 5645 and 5646, two warps complete the execution. At cycle 6731 and 6762, another 2 warps finish execution, leaving the active warps to be 2. The longest running warp is not done until cycle 24332. Unfortunately, due to TB-level resource management, all the warps in the TB will continue occupying their resources until cycle 24332. Therefore, the ratio of temporal resource underutilization due to TB-level resource management for TB0, i.e., RTRU, is high (49.7%).

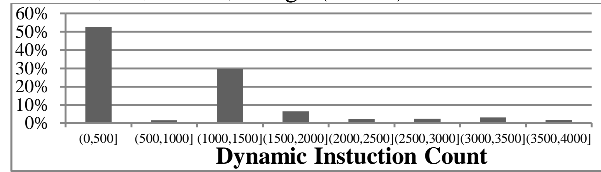

Figure 6. The histogram of per-warp dynamic instruction counts for the benchmark, RAY. The notation (a,b] means the range: {x|a<x<=b}.

3.2.2. Program-dependent workload imbalance. Another reason for workload imbalance among warps in a TB is due to the nature of programs/kernels. As discussed in Section 3.2.1, GPU kernels use the thread ids to determine the corresponding workload. It is quite common that some threads are assigned more workloads by design. Take the benchmark ST as an example, a segment of which is shown in Figure 7. The condition of the 'if' statement indicates that the threads with small ids along the *X* and *Y* directions have more work to do than those with large thread ids.

```
__global__ void block2D_reg_tiling(...) {
   int i = blockIdx.x*blockDim.x+threadIdx.x;
   int j = blockIdx.y*blockDim.y+threadIdx.y;
   if( i>0 && j>0 &&(i<nx-1) &&(j<ny-1) ) {
        …
}}
```

Figure 7. A code example from the benchmark ST

From the histogram of per-warp dynamic instruction counts shown in Figure 8, we can see that there are two types of warps in ST. The first has less than 50 instructions and the second has around 1250 instructions. It directly corresponds to the 'if' statement in Figure 7.

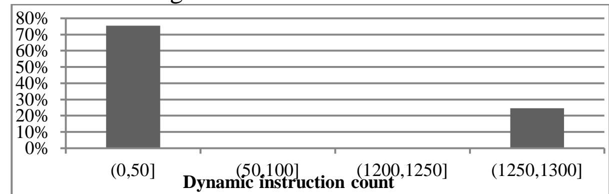

Figure 8. The histogram of per-warp dynamic instruction counts for the benchmark ST.

The code-dependent workload imbalance of ST leads to warp-level divergence as shown in Figure 9. As we can see from Figure 9, for TB0 of ST, during the first 500 cycles, all of its 32 warps are busy. At cycle 1663, the number of active warps drops to 8 and these 8 warps take long time (over 24000 cycles) to execute. Therefore, the resources reserved for the 24 early finished warps are wasted and the RTRU is as high as 70.1% for this TB. Across all the 64 TBs of ST, the average RTRU is 81.7%.

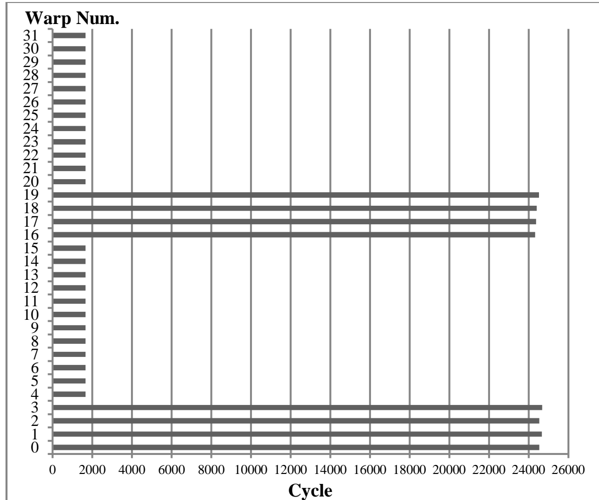

Figure 9. The lifetimes/execution times of different warps in TB0 of the benchmark ST. The ratio of temporal resource underutilization (RTRU) is 70.1%.

3.2.3. Memory divergence. Even when the workloads / dynamic instruction counts are identical for different warps in a TB, warp-level divergence can still happen due to run-time events such as cache hits/misses.

Taking the benchmark CT as an example, although each warp executes the same number of instructions, different warps encounter different numbers of cache misses, which in turn result in different execution times. Since CT uses texture memory for its input data, in Figure 10, we report the per-warp texture cache (T-cache) miss rate, measured in the number of misses per 1k instructions (MPKI), for TB (127, 23). We choose this TB rather than TB0 (or TB(0,0)) to make sure that the cache has been warmed up.

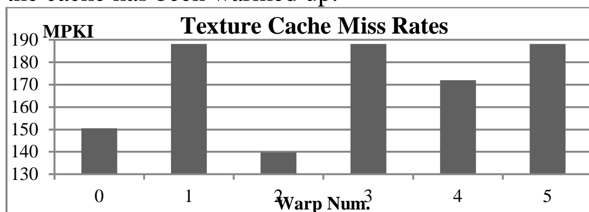

Figure 10. The texture cache miss rate, measured in MPKI, of the different warps in TB (127, 23) of CT.

As shown in the Figure 10, there is a significant variation in T-cache rates among the 6 warps in TB (127,23) of the benchmark CT. Warps 1, 3, 5 have high cache miss rates (188.2 MPKI) while warp 0 and 2 has much lower cache miss rates (150.5 and 139.8 MPKI). Such memory divergence leads to warp-level divergence as shown in Figure 11.

As we can see from Figure 11, the warps with higher T-cache miss rates usually have higher execution times. In many-core architectures like GPUs, the execution time is not directly proportional to the cache miss rates as TLP can hide a significant amount of miss latency. Nevertheless, for the TB (127, 23) of the benchmark CT, memory divergence results in a RTRU of 6.4%.

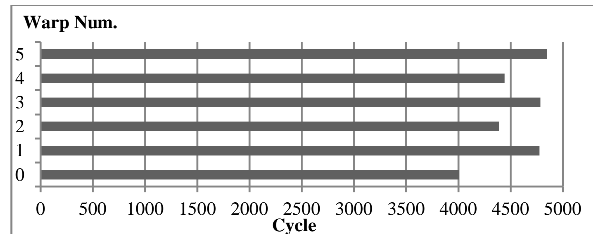

Figure 11. The lifetime/execution time of different warps in TB (127, 23) of the benchmark CT. The ratio of temporal resource underutilization (RTRU) is 6.4%.

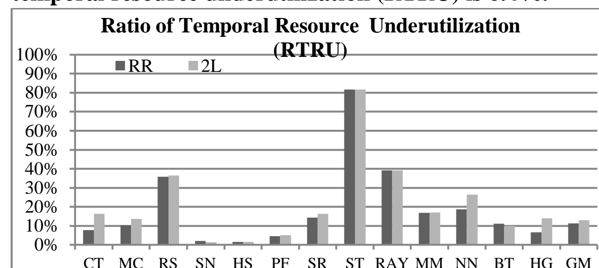

Figure 12. Warp-level divergence measured in the ratio of temporal resource underutilization (RTRU) for different warp scheduling policies.

3.2.4. Warp scheduling policy. Another reason for warp-level divergence is due to warp scheduling policies which may prioritize some warps over the others in a TB. For example, the recently proposed two-level scheduling [9][17] tries to better overlap memory access latency with computations by intentionally making some warps runs somewhat faster than others. In Figure 12, we compare the impact from two scheduling policies, round robin (labeled as 'RR') and two-level (labeled as '2L'). For either policy, we report the impact of the overall warp-level divergence (i.e., the warp-level divergence due to all the reasons discussed in Section 3) using the average RTRU across all TBs for each benchmark. As we can see from the Figure 12, the benchmarks, CT, MC, RS, PF, NN HG and SR, show higher warp-level divergence when the two-level scheduling policy is used. On the other hand, the benchmark SN shows slightly less warp-level divergence when the two-level scheduling policy is applied. On average, when the two-level scheduling policy is used, the RTRU is 13.0%. When the round-robin policy is used, the RTRU is 11.3%. Considering the overall impact of warp-level divergence, among the eleven benchmarks in our study, ST has the highest average RTRU of 81.7%. Seven benchmarks, MC, RS, SR, ST, RAY, MM, NN and BT, show significant warp-level divergence and their RTRUs are at least 10%. Such results highlight that significant resources are wasted due to warp-level divergence.

4. WarpMan: Warp-level Resource Management

To overcome both the spatial and temporal resource underutilization problems of TB-level resource management, we propose to manage the GPU resources at the warp granularity and we refer our proposed approach as WarpMan. The key idea is that rather than requiring the resource for a whole TB, as long as an SM has enough resource for one or more warps, it can be dispatched to the SM. In addition, once a warp in a TB finishes execution, its allocated resource can be released and can be used to accommodate a new warp.

Although TB-level resource management suffers resource underutilization, it simplifies shared memory management as shared memory data are shared among all the threads/warps in a TB. To ensure the correctness and not complicate shared memory management, we keep TB-level shared memory management intact while enabling our warp-level management for other resources. In other words, the shared memory is still allocated and released at the TB granularity while other resources are managed at the warp granularity. For example, the benchmark MM uses 2kB shared memory in a TB and each TB contains 256 threads (or 8 warps). Its register usage is 6144 registers per TB (i.e., 768 registers per warp). Given a register file of 32768 registers (or a 128kB register file) and 48 kB shared memory on an SM, one SM can run five TBs, which will use a total of 30720 registers and 10kB shared memory. Our proposed WarpMan will try to enable one more TB, although a 'partial' one, to run on the SM to fully utilize the registers. To do so, it first checks whether the SM has enough available shared memory for one additional TB. If so, the shared memory for one TB is allocated and this additional TB is dispatched to the SM. As there is sufficient available shared memory for a TB and the unutilized 2048 (=32768 – 30720) registers can support up to two more warps, one more TB is dispatched to the SM. However, as only part (i.e., 2 warps) of this TB is eligible for execution, this TB is referred to as a 'partially dispatched' TB to differentiate it from the TBs with full resource allocation. If the available shared memory is not sufficient for a new TB, no TB is dispatched to the SM. When a warp from the partially dispatched TB reaches a thread synchronization instruction, i.e., __syncthreads(), it simply stalls and waits for other warps, which may not have obtained the required resource on this SM yet, i.e., ineligible for execution. Note that a partially dispatched TB stopping at a __syncthreads() operation is still beneficial to performance. The reason is that the __syncthreads() operations often appear after long-latency operations such as loading global memory data to shared memory. Take the benchmark MM as an example, before the first __syncthreads() barrier, the global memory data are loaded into the shared memory. Therefore, a partially dispatched TB (i.e., a few warps) will load the global memory data before stopping at the __syncthreads() operations, thereby improving the performance. Due to data locality within TBs, the data loaded by those early issued warps can also benefit other warps in the same TB through L1/L2 caches. Furthermore, warp-level divergence may exist after the last synchronization point in a kernel. Therefore, releasing the resources of the early finished warps enables new warps to be dispatched more promptly.

To support WarpMan, we make few lightweight hardware changes for TB-level resource management. It contains two parts, (a) partial-TB dispatch logic in the TB dispatcher and (b) a small workload buffer in each SM to keep the information of a partially dispatched TB. Note that all these hardware units are only used at the TB dispatching and warp completion time. They do not affect SIMD execution pipeline.

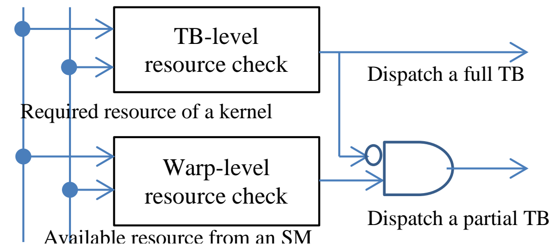

Figure 13. Thread-block dispatching logic.

4.1. Partial Thread Block Dispatch Logic

With TB-level resource management, before the GPU kernel dispatcher dispatches a TB to an SM, it checks both the available resources in the SM against the TB-level resource requirement. The following resources are checked:

- The available entries in the warp scheduler
- The number of available TBs
- The number of available registers
- The size of available shared memory

If all these available resources are sufficient for a TB, the dispatcher will send one TB to the SM. To support WarpMan, we append the TB-level resource check logic with a warp-level resource check logic unit as shown in Figure 13. When the TB-level resource check logic indicates that there is sufficient resource in the SM to host a full TB, a full TB is dispatched to the SM. Otherwise, the warp-level resource check logic is used to determine whether a partial TB can be dispatched to the SM. This way, the TB dispatcher logic will dispatch full TBs to an SM until there is not

enough resource for one more TB. At this time, a partial TB may be dispatched.

Same as the TB-level resource check logic, the warp-level resource check logic still checks the shared memory resource at the TB granularity. However, it checks the remaining resources at the warp granularity: if there is at least one available warp scheduler entry and at least one unoccupied TB (i.e., satisfying the limit on the maximal number of TBs on an SM), and if there are enough available registers for one or more warps, a 'partial' TB can be dispatched to the SM. Again, a partial TB here means that all the warps of the TB are dispatched to the SM, but not all of them are eligible for execution. Using the MM example, an SM can host five full TBs and a partial TB. In the partial TB, only two of its eight warps can start execution on the SM and the remaining six warps of the TB are referred to as dispatched but unissued warps. Within a partial TB, warps are selected for resource allocation and execution based on the order of warp ids.

When a warp finishes execution in an SM, its allocated resource, i.e., the registers and the warp scheduler entry, will be reclaimed. Then, the workload buffer will be checked first to see whether there are unissued warps. If the workload buffer is empty, the TB dispatching logic will be invoked to dispatch one more TB (full or partial) to the SM. This way, it guarantees that there will be at most 1 partial TB being dispatched to an SM, which avoids any potential deadlock problem due to partial TB dispatch. Also, as the TB dispatching logic still dispatched a TB and an SM at a time, all the threads of a TB are executed in the same SM as before. This property is necessary to support shared memory usage and synchronization among threads within a TB.

4.2. A Workload Buffer

Since our proposed TB dispatcher dispatches a partial TB to an SM even when the SM does not have enough resources for all the warps in the TB, we need to keep the information of such a partial TB. To do so, we propose a single-entry workload buffer, which contains five fields as shown in Figure 14. The reason for single entry is that there is at most one partial TB being dispatched to an SM. In the workload buffer, the 'P_TBid' field contains which physical TBid within the SM that the partially dispatched TB maps to. This field takes 3 bits assuming an SM can support up to 8 TBs. The 'L_TBid' field contains the logic TBid of the partially dispatched TB and it takes 26 bits as the maximum dimension of a grid is 1024x1024x64 [18]. The 'start_wid' and 'end_wid' fields store the warp ids for the first and last unissued warp. Either field takes 5 bits each as the maximum number of warps per TB is 32. The 1-bit 'valid' field marks whether all the warps in this partially dispatched TB are issued or not. Overall, this single-entry workload buffer takes 40 (=26+3+5+5+1) bits.

P_TBid	L_TBid	Start_wid	End_wid	valid

Figure 14. The workload buffer

When a partial TB is dispatched to an SM, the workload buffer stores the unissued warps and sets the valid bit as 1. When a warp in the SM finishes execution, its registers and its warp scheduler entry are released. If the valid bit of the workload buffer is 1, the warp with the warp id of 'start_wid' is issued from the workload buffer and the 'start_wid' field is increased by 1. When 'start_wid' is equal to 'end_wid', it means that all the warps of this partially dispatched TB have been issued. In this case, we reset the valid bit of the workload buffer.

Overall, our proposed WarpMan overcomes both the spatial and temporal resource underutilization problems of TB-level resource management. It makes better use of GPU resources and achieves higher TLP. Although higher TLP provides better latency hiding capability, it has been observed that increased TLP sometimes may hurt the performance due to the cache contention problem [22]. To address this performance anomaly, we proposed to either use the dynamic SM-dueling approach [14] or a simple static threshold to limit the number of active warps. More details are discussed in Section 6.1.

Note that our proposed WarpMan approach does not affect the warp scheduling policy except that it provides more candidates to be scheduled. In the case of multi-kernel execution, i.e., when an SM is capable of executing TBs from different kernels concurrently, the released resource from an early finished warp will be used only to accommodate a warp from the same kernel since all warps from the same kernel have the same resource requirement. This is essentially the same requirement for TB-level resource management for multi-kernel execution. In other words, WarpMan does not incur additional complexity to existing hardware for multi-kernel execution.

4.3. A Software Alternative

Another way to achieve warp-level resource management is to change the source code to reduce the TB size, e.g., to the warp size. At the same time, the hardware is changed to simply increase the maximal number of TBs that can run on an SM to a large number, e.g., 48 (= max. num. of threads in an SM/warp size). One limitation of this approach is that the small TB size limits data sharing and synchronization among the threads in a TB. In our study, seven benchmarks have shared memory usage and synchronization, which make it difficult to adjust the TB size. For the benchmarks without shared

memory usage or synchronization, we evaluate the performance impact of this approach in Section 6.3.

Table 1. The baseline GPU configuration.

Shader code frequency	1.4GHz
Number of SMs	15
Warp size	32
SIMD width	32
Max. num. of TBs per SM	8
Max. num. of threads per SM	1536
Scheduling policy	RR/ Two-level
Per-SM Register file	128kB (32k registers)
Per-SM Shared memory	48kB
Per-SM L1 D-cache	8-way set assoc. 64B cache block (16kB in total)
L2 Cache	8-way set assoc. 64B (256kb per MEM channel)
Number of MEM channels	6
GDDR Memory	16 banks, 924Mhz, total bandwidth: 173GB/S, TCL = 12, TRP = 12, TRCD = 12

5. Experimental Methodology

We modified the timing simulator GPGPUsim V3.2.0 [1] to evaluate the performance gains from our proposed schemes. Our baseline GPU configuration, modeled based on NVIDIA GTX480 GPUs [18], is shown in Table 1. Note that our proposed approach does not increase the sizes of the critical resources. Compared to the resources shown in Table 1, the hardware overhead of our proposed approach, including the TB dispatcher logic and the per-SM 40-bit workload buffer, is nearly negligible. The baseline warp scheduling policy is round robin (RR) and the two-level warp scheduling policy [17] is examined in our design space exploration in Section 6.3. In our design space exploration, we also vary the register file size and the SIMD width to evaluate the effectiveness of our approach in different configurations.

Table 2. The benchmarks used for evaluation.

Benchmarks	Shared Mem. Per TB	Warp Per TB	TB Per SM	Reg. Per TB	IPC
ConvolutionTexture(CT)	0	6	7	3068	814.1
MonteCarlo (MC)	0	12	4	7680	545.9
RadixSort (RS)	4k	16	2	11776	536.1
SortingNetworks (SN)	8k	16	3	8704	643.9
HotSpot (HS)	3k	8	3	8960	494.3
PathFinder (PF)	2k	8	6	3228	740.2
SRad (SR)	4k	16	3	2560	563.5
Stencil (ST)	0	32	1	17408	216.3
RayTracing (RAY)	0	6	2	12288	221.1
MatrixMultiply (MM)	2k	8	5	6144	420.4
Neural Network(NN)	0	8	5	6114	520.2
B+Tree(BT)	0	16	3	5632	410.5
Histogram(HG)	4k	16	3	10240	358.8

To measure the energy consumption impact, GPUWattch [12] is used together with GPGPUsim. The performance statistics as well as the hardware configuration parameters are passed from GPGPUsim to GPUWattch for power analysis.

We select 13 benchmarks from Nvidia CUDA SDK [19], the Rodinia benchmark suite [5], the Parboil benchmark suit [11], and GPGPUsim to cover a wide range of application domains. For each benchmark, we report the number of warps per TB, number of TBs per SM, the register usage per TB as well as the baseline performance measured with instructions per cycle (IPC) in Table 2. For each benchmark, we use the default input included in the benchmark suite.

6. Experimental Results

6.1. Performance Impact of WarpMan

In this experiment, we investigate the performance gains from our proposed WarpMan scheme and the results are shown in Figure 15. WarpMan addresses both temporal and spatial resource underutilization problems due to TB-level resource management. To isolate the performance impact, we first apply WarpMan to address only the temporal resource underutilization problem. In other words, we allow early finished warps to release their resource for new warps but we do not make use of the fragmented resources or the spatially underutilized resource. The performance of addressing only the temporal resource underutilization problem is labeled ‘temp’ in Figure 15. Then, we enable WarpMan to use the spatially underutilized resources to accommodate more warps, i.e., eliminating both the temporal and spatial resource underutilization, and the results are labeled ‘temp+spatial’ in Figure 15.

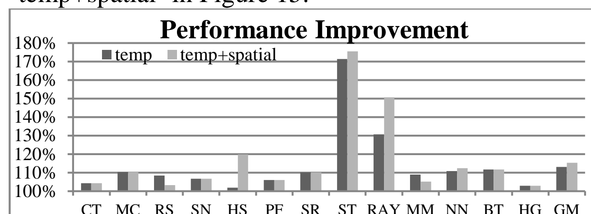

Figure 15. The performance improvements from our proposed approaches.

As reclaiming resources at the warp level mainly addresses temporal resource underutilization due to warp-level divergence, the performance gains of ‘temp’ correlate to the RTRU reported in Figure 12. With a RTRU of 81.7%, the benchmark ST enjoys an impressive performance gain of 71.4% from releasing the resources of the early finished warps. Both RAY and RS have significant RTRUs while RS has a higher number of concurrent warps in an SM (32) than RAY (12). Therefore, the performance improvement for RAY (30.1%) is higher than that for RS (8.4%). On average, our ‘temp’ approach achieves a 13.0% speedup and we do not observe any slowdowns for the ‘temp’ approach.

When we combine 'temp' with 'spatial' to address both temporal and spatial resource underutilization issues, additional performance gains are achieved for the benchmarks HS, ST, and RAY. For HS and RAY, their numbers of TBs that can run on an SM concurrently are limited by the register file size. With our proposed approach, a partial TB can make effective use of such otherwise idle resource to improve TLP and improve performance. On average, our proposed 'temp+spatial' achieves 15.3% performance improvements over the baseline. For applications with __syncthreads(), such as RS, SN, HS, PF, SR and MM, both 'temp' and 'temp+spatial' still achieve significant performance improvement even though those partially dispatched TBs have to stall at the first _syncthreads().

We also observed that, for benchmarks, RS and MM, utilizing spatially underutilized resource (i.e., 'temp+spatial') indeed increases TLP but results in performance loss compared to the 'temp' approach alone. The reason is due to the contention problem resulting from increased active warps. For MM, the problem is the cache contention: the L1 D-cache miss rate is increased from 57.7% to 60.7%. For RS, the contention happens at the row buffers in off-chip DRAM. The 'temp' approach has much higher row buffer locality (24.96 accesses per activation) than 'temp+spatial' (10.74 accesses per activation).

To address the contention problem, we propose two different approaches. The first uses SM dueling. One SM is always equipped with 'temp+spatial' while another SM always equipped with 'temp'. During run time, we compare the IPC of these two competing SMs, and decide whether to turn on/off 'spatial' resource allocation. The second approach simply sets a static threshold to determine whether there are enough concurrent warps running. Our empirical results indicate a reasonable threshold as 36, i.e., once the number of concurrent warps is equal to 36, additional warp-level allocation is disabled.

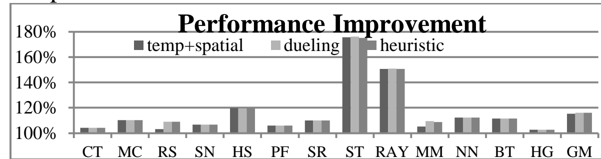

Figure 16. The performance improvement from the SM-dueling and the static threshold (labeled 'heuristics') approaches to eliminate the contention problem.

The performance results of these two approaches, labeled as 'dueling' and 'heuristic' are shown in the Figure 16. As seen from the figure, both approaches effectively eliminate the performance degradation caused by contention. On average, when the contention problem is eliminated, we achieve 16.0% performance gains over the baseline.

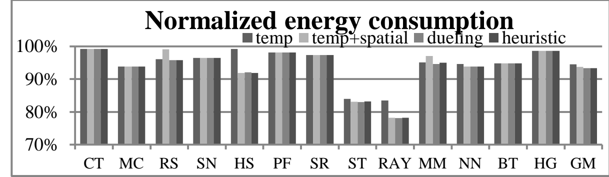

Figure 17. The total energy savings from our proposed approaches.

6.2. Energy Savings from WarpMan

The strong performance gains from our proposed WarpMan approach directly translates into significant static energy savings. Since our approaches do not change the number of ALU/Memory operations, there is little change in dynamic energy consumption. In Figure 17, we report the total energy savings achieved by our techniques. For applications with high performance gains, such as HS, RAY and ST, the energy savings are significant 8%, 16.8% and 21.7%, respectively. On average, we achieve 5.4% total energy reduction when only addressing the temporal resource underutilization, 6.2% when eliminating both temporal and spatial resource underutilization, 6.7% when applying either the dueling or heuristic approach to eliminate the contention problem.

6.3. Comparison to the Software Approach for Warp-Level Resource Management

As discussed in Section 4.3, to achieve warp-level resource management, we can reduce the TB size to 32 and increase the number of concurrent TBs on each SM to 48. Note that this represents a selection for fine-grain TB size, larger TB sizes are also possible. Among our benchmarks, seven of them, RS, SN, HS, PF, SR, and MM, have shared memory usage and __syncthreads() functions, which make them unfit for this approach. Therefore, we focus on the remaining six benchmarks. The performance results, labeled as 'TBsize_32', are shown in Figure 18. The results of our proposed WarpMan scheme, labeled as 'temp+spatial', are included for comparison. From Figure 18, we can see that the software approach can achieve similar performance to WarpMan for RAY and MC. The benchmark Ray achieves higher performance using the software approach due to a '__syncthreads()' function in the kernel, which leads to some stall cycles for WarpMan. For NN and BT, the code is written such that one TB processes 1 image (or one query). If we reduce the TB size, we have to either increase the workload of each thread or use multiple TBs to process 1 image (or 1 query), which require non-trivial code change and the resulting performance is even lower than the baseline.

For CT and ST, the warps within the original-sized TB have strong data reuse. If we reduce the TB size to

32, adjacent TBs may be dispatched to different SMs. For CT, we observe that the texture cache miss rate increases from 3.8% to 41.5% when the TB size is reduced from 192 to 32. Therefore, the IPC drops from 814.1 to 106.3. In comparison, our WarpMan does not alter cross-warp data reuse and further improves the IPC to 848.0. The benchmark ST has high warp-level divergence. Therefore, the benefits of reducing the block size overweigh its adverse impact on data reuse.

On average, our proposed WarpMan achieves a 25.1% performance improvement while the software approach actually leads to a 47.9% slowdown.

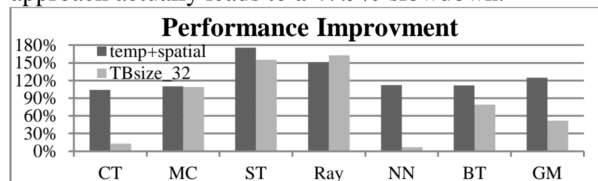

Figure 18. Performance comparison between the software approach (i.e., small TB sizes) and WarpMan

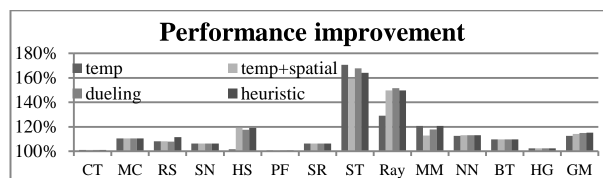

Figure 19. The performance improvement from our approaches using the two-level warp scheduling policy.

6.4. Design Space Exploration

Since the scheduling policy may impact warp-level divergence, in the next experiment, we evaluate the impact of the two-level scheduling policy on our proposed approach. As we can see from Figure 19, our proposed approaches are effective for two-level warp scheduling and achieve a 14.1% performance improvement on average when both 'temp' and 'spatial' are used. Similar to what we observed in Section 6.1, increasing TLP does not always improves performance due to the contention problem. With either the SM-dueling or the static threshold approach (the same threshold 36 is used here as well), we can eliminate the contention problem and recover the lost performance gains. On average, with these two approaches, we achieve 15.0% and 15.3% performance improvement, respectively.

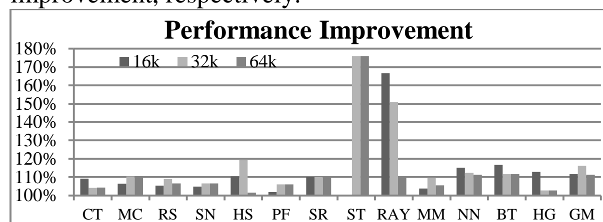

Figure 20. The performance improvement from WarpMan for different register file sizes

In the next experiment, we vary the register file size from 16k registers, 32k registers, to 64k registers and evaluate the performance gains from our proposed WarpMan approach. Set-dueling is used to eliminate the contention problem in this experiment. The results are shown in Figure 20. Among the benchmarks, ST needs 17408 registers for each TB. Therefore, we do not have the performance results shown in Figure 20 for ST when the register file size is 16k registers.
As discussed in Section 3.1, a large register file enables the register file sensitive benchmarks to achieve higher TLP. Therefore, for the benchmarks such as CT, NN, BT, HG and RAY, the benefits from WarpMan are higher when the register file size is 16k registers and lower when the register file size increases, as shown in Figure 20. For the benchmark SN, PF, BT, HG and SR, increasing the register number to 64k has no performance impact because their TLPs are not limited by the register file size as shown in Figure 12 and Figure 15. For the ST and MC, since the TLP is limited by both the registers and the warp scheduler size (i.e., the maximal number of threads), so a larger register file (64k) alone will not enable more TBs to be dispatched to an SM. Therefore, the effectiveness of our approach does not alter when increasing the register number from 32k to 64k. For MM, row buffer locality plays an important role besides TLP. Therefore, the performance trend does not directly correlate to the TLP improvement. On average, for a GPU with a 256kB register file (64k registers) in an SM, our proposed approach achieves a performance improvement of 11.6%. For a small register file of 16k registers, our proposed approach achieves a performance improvement of 11.3% on average.

We also vary the SIMD width to investigate the performance benefit of our proposed WarpMan scheme. The results show that on average, it achieves 16.8% and 16.1% performance improvement for the SIMD width of 8 and 16, respectively.

Overall, these experiments show that our proposed schemes are effective for different GPU configurations.

7. Related Work

The resource fragmentation or spatial resource underutilization problem due to TB-level resource management has been observed in [13]. It has been leveraged in [15] to turn-off un-utilized registers to reduce static energy consumption.

Resource underutilization due to branch divergence or thread-level divergence has been well studied [2][4][6][7][8][10][16][20][21] and the problem is due to SIMD style execution. In our work, we make a new observation that different warps in the same TB can have quite different execution times and we call this behavior as warp-level divergence. The fundamental

reason for resource underutilization, however, is due to TB-level resource management.

Shared memory multiplexing [26] targets at the shared memory management and is complementary to our proposed WarpMan scheme. As discussed in Section 4, WarpMan still allocates and releases at the TB granularity while managing the register file and the warp scheduler at the warp granularity. With dynamic shared memory allocation and deallocation proposed in [26], the allocated shared memory can be released earlier than the TB completion time. This way, WarpMan can be more effective for the benchmarks whose TLP is constrained by their shared memory usage as well.

As discussed in Section 2, a virtual register table is used to dynamically manage the physical register file [23], which enables more concurrent TBs on an SM if the resource bottleneck is the register file. Our proposed WarpMan can be used together with this approach to make more efficient use on both the virtual register table and the physical register file. In other words, a warp releases its occupied virtual register table entries immediately after it finishes execution such that these virtual register table entries can be used for a new warp.

8. Conclusions

In this paper, we highlight that TB-level resource management may lead to severe resource underutilization. First, TB-level resource allocation leads to resource fragmentation and some parts of the resource are never utilized. Second, TB-level resource release and reallocation suffers from warp-level divergence since different warps in the same TB may have significantly different execution time and the resources allocated to the early finished warps are essentially wasted.

Our proposed solution is warp-level resource management. Rather than requiring the resources, including the register file and the warp scheduler entries for a whole TB, a partial TB can be dispatched to an SM if it has the resource for one or more warps. During execution, as soon as a warp finishes execution, its allocated resources are released and can be used to accommodate a new warp. We present our lightweight architectural support for warp-level resource management. Our experimental results show that significant performance gains, up to 76.0% and 16.0% on average, and energy savings, up to 21.7% and 6.7% on average, are achieved.

9. Acknowledgement.

We thank the anonymous reviewers for their insightful comments to improve our paper. This work is supported by an NSF project 1216569, an NSF CAREER award CCF-0968667 and a grant from the DARPA PERFECT program.

10. References

[1] A. Bakhoda, et al., "Analyzing CUDA Workloads Using a Detailed GPU Simulator," ISPASS-2009, 2009

[2] N. Brunie, et al., "Simultaneous Branch and Warp Interweaving for Sustained GPU Performance" ISCA-39, 2012.

[3] CUDA programming guide

[4] J. D. Collins, et al., "Control flow optimization via dynamic reconvergence prediction," MICRO-37, 2004

[5] S. Che, et al., "Rodinia: A Benchmark Suite for Heterogeneous Computing," IISWC-2009, 2009.

[6] G. Diamos, et al., "SIMD Re-Convergence at Thread Frontiers," MICRO-44, 2011.

[7] W. W. Fung, et al., "Thread Block Compaction for Efficient SIMT Control Flow," HPCA-17, 2011.

[8] W. Fung, et al., "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow," MICRO-40, 2007.

[9] M. Gebhart, et al., "Energy-efficient mechanisms for managing thread context in throughput processors," ISCA-38, 2011

[10] T. D. Han, et al., "Reducing Branch Divergence in GPU Programs," GPGPU-4, 2011.

[11] IMPACT Research Group.The Parboil Benchmark Suite.

[12] J. Leng, et al., "GPUWattch: Enabling Energy Optimizations in GPGPUs, " ISCA-40, 2013.

[13] D. Kirk, et al., "Programming Massively Parallel Processors A hand-on Approach Morgan Kaufmann, 2012.

[14] J. Lee et al TLP-Aware Cache Management Schemes for a CPU-GPU Heterogeneous Architecture HPCA-18, 2012.

[15] A. Mohammad, et al.,"Warped Register File: A Power Efficient Register File for GPGPUs," HPCA-19, 2013.

[16] J. Meng et al.,"Dynamic Warp Subdivision for Integrated Branch and Memory Divergence Tolerance" ISCA-37, 2010.

[17] V. Narasiman, et al.," Improving GPU Performance via Large Warps and Two-Level Warp Scheduling," MICRO-44, 2011.

[18] NVIDIA. Fermi: NVIDIA's Next Generation CUDA Compute Architecture, 2011.

[19] NVIDIA. CUDA C/C++ SDK Code Samples, 2011. http://developer.nvidia.com/gpu-computing-sdk, 2011

[20] Minsoo Rhu, et al.,"The Dual-Path Execution Model for Efficient GPU Control Flow," HPCA-19, 2013.

[21] Minsoo Rhu, et al.,"CAPRI: Prediction of Compaction-Adequacy for Handling Control-Divergence in GPGPU Architectures," ISCA-39, 2012.

[22] T. Rogers, et al.,"Cache-Conscious Wavefront Scheduling," MICRO-45, 2012.

[23] D. Tarjan, et al., "On demand register allocation and deallocation for a multithreaded processor." US Patent 20110161616 A1. 2009

[24] J Tan et al RISE: Improving Streaming Processors Reliability against Soft Errors in GPGPUs PACT-21, 2012

[25] J. Tan, et al., "Analyzing Soft-Error Vulnerability on GPGPU Microarchitecture," IISWC-2011, 2011.

[26] Y. Yang, et al., "Shared Memory Multiplexing: A Novel Way to Improve GPGPU Throughput," PACT-21, 2012.