Enabling Efficient Preemption for SIMT Architectures with Lightweight Context Switching

Zhen Lin
Dept of Electrical and Computer Engineering
North Carolina State University
Raleigh, North Carolina
zlin4@ncsu.edu

Lars Nyland
NVIDIA Corporation
Durham, North Carolina
lnyland@nvidia.com

Huiyang Zhou
Dept of Electrical and Computer Engineering
North Carolina State University
Raleigh, North Carolina
hzhou@ncsu.edu

***Abstract*—Context switching is a key technique enabling preemption and time-multiplexing for CPUs. However, for single-instruction multiple-thread (SIMT) processors such as high-end graphics processing units (GPUs), it is challenging to support context switching due to the massive number of threads, which leads to a huge amount of architectural states to be swapped during context switching. The architectural state of SIMT processors includes registers, shared memory, SIMT stacks and barrier states. Recent works present thread-block-level preemption on SIMT processors to avoid context switching overhead. However, because the execution time of a thread block (TB) is highly dependent on the kernel program. The response time of preemption cannot be guaranteed and some TB-level preemption techniques cannot be applied to all kernel functions. In this paper, we propose three complementary ways to reduce and compress the architectural states to achieve lightweight context switching on SIMT processors. Experiments show that our approaches can reduce the register context size by 91.5% on average. Based on lightweight context switching, we enable instruction-level preemption on SIMT processors with compiler and hardware co-design. With our proposed schemes, the preemption latency is reduced by 59.7% on average compared to the naive approach.**

I. Introduction

State-of-the-art SIMT processors or GPUs exploit high degrees of thread-level parallelism (TLP). As a side effect, SIMT processors feature high amounts of on-chip resources to accommodate the contexts of the large numbers of concurrent threads. For example, in the NVIDIA GK110 (Kepler) architecture, each stream multiprocessor (SM) has a 256KB register file and up to 48KB shared memory. Such large contexts result in long latency for context switching (meaning switching in a new kernel rather than switching among the running threads/warps, which SIMT processors support natively). To reduce the overhead, TB-level context switching techniques, SM-draining [27] and SM-flushing [22] have been proposed. The key idea of SM-draining is to wait for all running TBs on an SM to finish, then to launch the TBs from the new incoming kernel to the SM. The drawback of this solution is that the preemption latency can be very high. In the worst scenario, a TB can have a lifetime as long as the kernel [12] [15], and the kernel may not be preempted at all. SM-flushing flushes the running TBs and then launches the new kernel. The limitation is that only kernels which conform the idempotent (re-executable) condition [8] can be preempted in this way. Also, the useful work is wasted when a running TB is flushed. To overcome such limitations, an integrated solution [22] is proposed based on the progress of a TB. If it is close to the end, TB draining is used. If it just begins execution, TB flushing is employed instead. In other scenarios, the baseline context switching, i.e., swapping the thread contexts, is performed.

In this paper, we propose novel ways to reduce and compress SIMT processor contexts to enable lightweight context switching. Three approaches are proposed. First, based on the observation that for some applications, the on-chip resource is significantly underutilized, we propose in-place context switching, which means that not all resources need to be released/spilled to accommodate a new kernel. Second, liveness analysis is used to exclude dead registers so as to reduce the register context sizes. In this paper, we observe the liveness of a vector register is dependent on the thread divergence. So the traditional liveness analysis algorithm is augmented for the SIMT architecture. Third, based on register pattern analysis, register contexts can be further compressed. The register pattern is explored in both warp-level and TB-level. These techniques can greatly reduce the context size that needs to be swapped to/from off-chip memory.

Based on the lightweight context switching, we use compiler and hardware co-design to enable instruction-level preemption for SIMT processors. The compiler analyzes the native assembly code to figure out the appropriate points for preemption. We introduce two new preemption instructions to annotate the preemption points, meaning that preemption is only enabled at this point. The preemption instruction checks the interruption signal and becomes a nop if there is no such a signal. If there is an interrupt signal, the context switching is handled by a special hardware pipeline to reduce and compress the architectural states. At last, the compressed states are saved to global memory. To restore the kernel, the context will be loaded from global memory. After decompressing, the architectural states are restored on the processor.

Besides preemption, our proposed lightweight context switching can also be used for long running applications on supercomputers. The reason is that long running applications on supercomputers are error prone. Therefore, checkpointing

SC16; Salt Lake City, Utah, USA; November 2016
978-1-4673-8815-3/16/$31.00

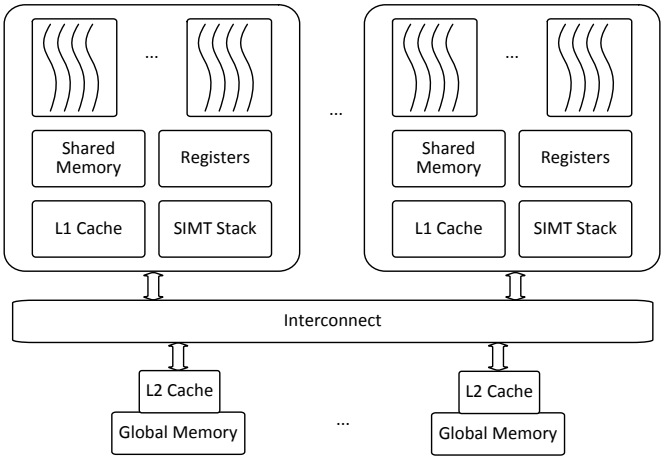

Fig. 1: Baseline SIMT processor architecture.

```
void kernel(float *A, float *B,
            int *Ahead, int *Bhead, int N) {
    int in_id, out_id;
    while ((in_id = atomic_inc(Ahead)) < N) {
        float in_data = A[in_id];
        float out_data = do_work(in_data);
        out_index = atomic_inc(Bhead)
        B[out_index] = out_data;
    } }
```

Fig. 2: Kernel code of persistent threads.

mechanisms are commonly used such that the supercomputer can resume from a prior checkpoint in the case of an error. Our proposed scheme enables efficient context saving, i.e., efficient checkpointing of GPU contexts.

We evaluate our context switching enabled preemption approach with the Rodinia [6] benchmarks. Our experiments show that the register context size can be reduced by 91.5% and the preemption latency can be reduced by 59.7% on average with our proposed lightweight context switching.

In summary, this paper makes the following contributions.

- We propose three techniques, in-place context switching, register liveness analysis, and register value compression to achieve lightweight context switching.
- A compiler and hardware co-design is proposed to enable instruction-level preemption for SIMT processors.
- We show that our proposed approach achieves low preemption latency.

The rest of the paper is organized as follows. Section II describes the SIMT architecture and motivates the proposed ideas. Section III presents the techniques for lightweight context switching. Section IV makes use of lightweight context switching for efficient instructional-level preemption. Section V reports the methodology and experiment results. Section VI discusses the related work and Section VII concludes.

II. Background and Motivation

A. Baseline Architecture

Figure 1 shows the baseline SIMT processor or GPU architecture. A SIMT processor is composed of a number of streaming multiprocessors (SMs). The SMs share a multi-banked L2 cache. Typically, one or more L2 banks are backed up with a memory controller to communicate with off-chip memory. The SMs and multiple L2 banks communicate through a crossbar or an interconnect network. The on-chip memory in each SM includes shared memory, the register file and L1 D-cache. The basic execution unit in SIMT processors is a warp. A warp is a collection of threads that run in the single-instruction multiple-data (SIMD) style. Each warp has a private space in the register file. A per-warp SIMT stack keeps track of the program counters (PCs) of the threads when a divergent branch is encountered [9]. One or more warps constitute a thread block (TB). All threads in one TB can synchronize and share data through shared memory. The threads in the same TB must be executed on the same SM and one SM can accommodate one or more TBs depending on the resource requirement of a TB.

When a kernel is launched, the resource requirement of a TB is provided to the SIMT processor. Based on its available resource, an SM decides whether one more TB can be dispatched to it. There are four types of resources that can limit the number of concurrent TBs on an SM: the register file, shared memory, the warp scheduler, and the TB slots. For example, in the NVIDIA GT200 architecture, the register file size is 128KB, the shared memory size is 48KB, and there are 48 warp scheduler slots (i.e., up to 48 warps can run concurrently) and 8 TB slots on each SM. For a kernel with 8 warps (i.e., 256 threads) in each TB, if each warp takes 3KB register space and each TB takes 8KB shared memory space, the maximum TB per SM is 5 as limited by the register file size. A warp/TB will hold the resources during its whole lifetime. The resources will be released only after it finishes execution.

B. Prior Preemption Techniques for SIMT Processors

The large context size on SIMT processors leads to high preemption overhead. To avoid the overhead, Tanasic et al. [27] proposed the SM-draining technique, in which all current running TBs need to finish before releasing the SM for the new kernel. However, the SM-draining technique can cause long preemption latency. In the experiments from Park et al. [22], the preemption latency of SM-draining can be as high as tens of milliseconds. A more extreme case, as shown in Figure 2, is a persistent kernel [12] [15]. In this kernel, TBs only exit when all the input elements are processed. In other words, TBs may have the lifetime as long as the overall kernel execution time. Such kernels cannot be preempted with the SM-draining technique.

To address the long preemption latency, Park et al. [22] proposed SM-flushing. Taking advantage of idempotent regions, the execution of the kernel can be stopped immediately and all intermediate results can be flushed. The kernel can be resumed

by relaunching the flushed TBs. SM-flushing only works on idempotent kernels, which means each TB can be re-executed multiple times without affecting the results. This is a strict limitation. Although Park et al. [22] also proposed relaxed idempotent conditions, the scheme does not work on certain cases. For example, for the kernel shown in Figure 2, in each iteration, the variables Ahead, Bhead and an element in B will be modified. So it cannot be safely flushed. Moreover, when flushed, all the progress made on the TB is wasted.

Because the SM-draining latency may be too long and SM-flushing wastes useful work, Park et al. [22] also used context switching for preemption. But the naive approach to swap all the occupied registers and shared memory incurs high overhead. For example, in the benchmark HS, the context size for each SM is about 140KB. For GTX480 with 15 SMs and the global memory bandwidth of 177GB/s, even if the global memory bandwidth is fully utilized, it would take at least 12 us to store such a large context. As pointed out in prior works [27], the SMs are completely underutilized during context save and restore.

III. Efficient Context Switching

For context switching, in order to properly restore a warp or TB, its architectural states must be preserved. For a warp, the architectural state includes its registers and SIMT stack. The SIMT stack contains thread execution information in the case of divergent branches and also includes the program counters (PCs). For a TB, besides the contexts of all its warps, the architectural state also includes shared memory and barrier states, keeping the information on which warps have reached a barrier and are waiting for others. The SIMT stack and barrier states tend to be very small compared to registers and shared memory. For the SIMT stack, each entry has three 32-bit registers, which are the next PC, active mask and reconvergent PC [9]. Based on the observation by Rhu et al. [24], the maximum stack depth is limited (11). So the maximum stack size is relatively small (132B). For barrier states, each barrier only needs 1 bit for each warp to record whether it has reached the barrier. Therefore, we focus on the context of registers and shared memory in this work. Using the BP_1 benchmark as an example, each thread has 13 registers (4B each) and each TB has 256 threads and 1128B shared memory. So the context size is 1664B per warp and 14440B per TB. Our goal is to make such context sizes much more manageable.

Next, we propose three complementary schemes, (a) in-place context switching to leverage unused resource, (b) register liveness analysis for architectural state reduction, and (c) value locality detection for architectural state compression.

A. In-Place Context Switching

As different applications exhibit different resource requirements, the fixed-size resources on SIMT processors are commonly underutilized. In Figure 3, we report the occupancy of both the register file and shared memory for different benchmarks. We can see that for most benchmarks, either (or both) type(s) of the resources is under-utilized. Take BP_1 as

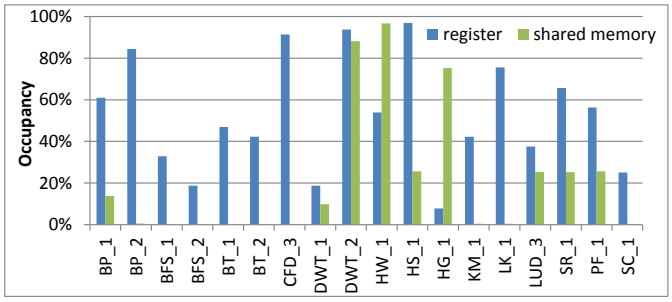

Fig. 3: Occupancy of the register file and shared memory.

an example, the register file occupancy is 60.9% and shared memory occupancy is 13.7% as the occupancy is limited by the number of threads (or the maximal number of warps). Such resource under-utilization have also been observed in prior works [2] [10]. In this paper, we make use of such unused resource to store the context of the warps/TBs to be switched out.

On the baseline SIMT architecture, when the warps of a thread block are dispatched to an SM, their logic registers are mapped to physical registers. In this paper, an allocation table is used for managing the register allocation. Each launched TB reserves one entry in the table to denote the start address and allocation size of the register file. When preemption occurs, the old kernel de-allocates the minimum number of TBs to make enough space for the new kernel. The remained TBs will keep reserving the register file. Such in-place context switching reduces the amount of data to be spilled and restored and enables fast preemption between two kernels. A similar but separate allocation table is used to manage the shared memory allocation.

Figure 4 (a) shows the register and shared memory allocation table when kernel K1 is running. K1 has 3 TBs on one SM, each TB allocates 300 vector registers and 8KB shared memory. In Figure 4 (b), K1 is preempted by K2, which launches 2 TBs per SM and each TB occupies 200 vector registers and 10KB shared memory. There are 1024 vector registers and 48KB shared memory in each SM on our baseline architecture. In this case, the register file deallocates and spills 2 TBs of K1 to accommodate K2, whereas none of shared memory needs to be deallocated.

In our implementation, each entry of register/shared memory table is 5B and the capacity for each of the tables is 16 entries. So the total overhead for the allocation tables is 160B.

A more aggressive option is to reallocate the dead register for the new kernel as proposed by H. Jeon et al. [13]. But such mechanism will increase the hardware complexity by introducing a register renaming table.

B. Architectural State Reduction

We propose to use liveness analysis to reduce architectural register states. Liveness analysis reports that at any program point, which registers are defined and may be potentially used before the next re-define. Only the values in live registers

Register allocation table

Kernel	Start Reg #	Size
K1_TB0	0	300
K1_TB1	300	300
K1_TB2	600	300

Shared memory allocation table

Kernel	Start Addr	Size
K1_TB0	0	8KB
K1_TB1	8KB	8KB
K1_TB2	16KB	8KB

(a)

Kernel	Start Reg #	Size
K2_TB0	0	256
K2_TB1	256	256
K1_TB2	600	300

Kernel	Start Addr	Size
K1_TB0	0	8KB
K1_TB1	8KB	8KB
K1_TB2	16KB	8KB
K2_TB0	24KB	10KB
K2_TB1	34KB	10KB

(b)

Fig. 4: Register and shared memory allocation tables before and after K1 is preempted by K2. (a) Before. (b) After.

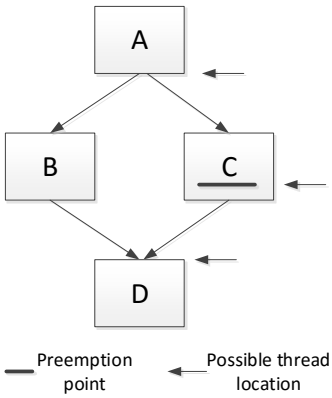

Fig. 5: Possible thread locations when a preemption point is reached.

need to be saved during context switching. At compile time, the compiler identifies live vector registers at each instruction and saves the results into a liveness table. Each entry in the liveness table corresponds to one static instruction and the register liveness information is encoded into a bit vector.

One option to provide the liveness bit vector at runtime is to load the liveness table to the SIMT processor when the kernel is launched. At any point of execution, liveness registers can be looked up by the PC of a thread. The problem of such fine-granularity approach is that to store liveness table may take huge hardware resource. In our baseline architecture, the maximum register number is 64, meaning each liveness entry is 8B. For a program with 1K instructions, liveness table will take 8KB storage on hardware. To avoid the overhead of liveness table, we choose selective points to enable preemption. In other words, instead of enabling preemption for each instruction, we only enable preemption at certain selected program points. At each preemption point, a preemption instruction is inserted with encoded liveness bit vector. At runtime, the liveness bit vector is fetched to the instruction buffer. The details of selective preemption is discussed in Section IV-A.

In our approach, an entire warp will be stopped and handled by the preemption handler if any thread has reached the preemption point. In the case of thread divergence, the liveness of a whole warp may be different with the threads which

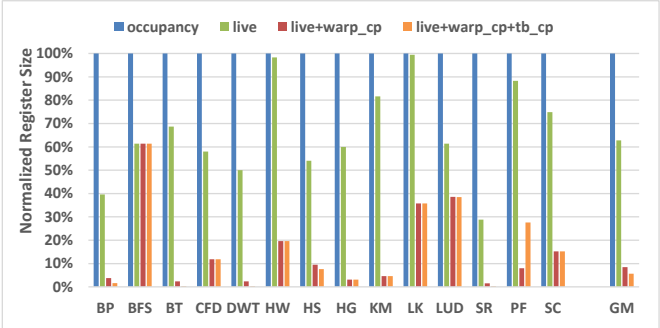

Fig. 6: Normalized register context sizes after liveness analysis and compression.

reached the preemption point. As shown in Figure 5 are the possible thread locations when a preemption point is reached. For example, one warp has two threads, T1 and T2. Assume that T1 and T2 diverge at basic block A, T1 executes path B and T2 executes path C. When T2 reaches the preemption point at path C, T1 can be either at the divergence point (end of A) or the reconvergence point (start of D). In this case, the live vector register for the preemption point should be the union of all these 3 possible thread locations.

In our compiler, we firstly perform the traditional liveness analysis without considering thread divergence. Then the divergence and reconvergence points are analyzed based on immediate post-dominator [9]. At last, the compiler calculates the union of liveness at the original preemption point, the divergence point and recovergence point. Because the thread divergence can only be determined at runtime, both liveness vector versions are saved. When handling the preemption of a warp, the SIMT stack will be checked to determine whether there is a divergence. If there is, the union version is used. Otherwise, the original version is used.

To evaluate the effectiveness of liveness analysis, we count the number of live registers in the Rodinia benchmarks at runtime. When a warp reaches a preemption point, the total liveness number is accumulated. Then, the sum is divided by the number of warps. The result is shown in Figure 6. Take BP_1 as an example, only 39.6% of occupied registers are live ones. Therefore, the average per-warp context size is reduced from 1664B to 656B. On average across all the kernels, 34.3% of the register context size can be reduced with liveness analysis.

C. Architectural State Compression

Register state compression is based on the observation that many register values in GPU programs conform certain patterns. S. Collange et al. [7] reports that uniform and strided are common patterns for GPU vector registers. A uniform register is defined as all scalar registers in a vector register have the save value, i.e. $V_i = a$. A strided register is defined as the scalar registers in a vector register conform arithmetic progression, i.e. $V_i = ai+b$. In this paper, we show that the TB dimension is an important factor for analyzing GPU register

```
for (i = 1; i <= LOG_H; i++) {
    int pow = __powf(2, i);
    if (ty % pow == 0) {
        float tmp = s_weight[ty+pow/2][tx];
        s_weight[ty][tx] += tmp;
    }
    __syncthreads();
}
```

Fig. 7: A kernel code snippet of BP_1.

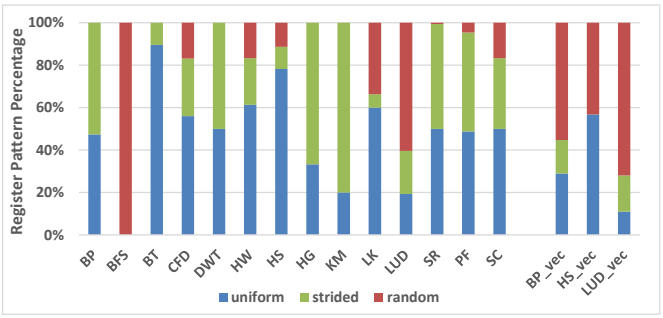

Fig. 8: Warp-Level register value locality analysis.

pattern. Also, we explore TB-level register compression to further exploit inter-warp data locality.

Warp-Level Compression

Warp-level register compression is used to leverage intra-warp value locality to compress vector registers. Take the kernel code snippet of BP in Figure 7 as an example. BP_1 has 16x16 TB dimension, tx and ty are the thread ids in the X and Y dimension, respectively. The variable i is uniform across all threads in a warp, and so are the variable pow and the base addresses of array s_weight. However, for a warp with 32 threads, the values of ty for warp 0 is "0, 0, ..., 0, 1, 1, ..., 1". The uniform pattern occurs for 16 scalar registers instead of the whole vector register. A similar situation happens for tx, which has the values "0, 1, ..., 15, 0, 1, ..., 15" for a warp. So, in this paper, the register pattern analysis is performed at the granularity of a pattern analyzing group (PAG). PAG is the minimum between the vector register width and the lowest TB dimension. In our baseline architecture, the vector register width is 32. So the maximum value of PAG is 32. When one of the TB dimensions is less than 32, PAG is the lowest TB dimension.

For vector registers containing uniform values, we can compress it into 1 scalar register. A strided vector register can be compressed to 2 scalar ones, i.e. a base and a stride. We refer to other registers as random ones, such as tmp in Figure 7.

To analyze the warp-level register value locality, we also take samples in the Rodinia benchmarks at runtime. For each sample, we count how many vector registers are uniform, strided or random in each warp. We only analyze the live registers. After execution, the average of all the samples is calculated. Figure 8 shows the result. For BP, 47.4% of its

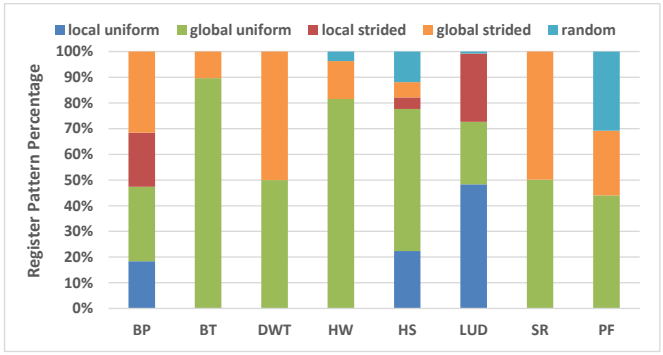

Fig. 9: TB-Level register value locality analysis.

live registers are uniform and 52.6% live registers are strided. Therefore, the per-warp context size can be further reduced to 64B on average. From Figure 6, we can see that on average 91.5% register context size can be reduced with combined liveness analysis and warp-level register compression.

In our benchmark, BP, HS and LUD has two-dimensional TBs and the PAG is different with the vector register width. In Figure 8 we use two approaches for warp-level register pattern analysis. The default approach is to use PAG as the analysis width whereas the BP_vec, HS_vec and LUD_vec use the vector register width, i.e., the warp size, as the analysis width. From the result we can see that our approach can exploit uniform and strided registers more effectively.

Figure 10 shows the logic design for register state compression. The inputs are PAG scalar registers. There are PAG-1 subtractors to calculate the differences between two adjacent values. The first subtraction result will be converted to a Boolean signal a, showing whether the difference is 0. The comparator takes the results of all subtractors and outputs 1 if all the results are equal, 0 otherwise. The output of the comparator is marked as signal b. This way, the signals a and b encode the value pattern, 01: uniform, 11: stride, 10/00: random. This logic is fully pipelined. Because the vector register width is 32 in our baseline architecture, the maximum of PAG is 32. In our experiments, the compression latency is assumed as 2 cycles. Such assumption is also used in a similar design for register compression [17].

The decompression process is relatively straightforward. For uniform registers, the value is duplicated PAG times for PAG registers in one warp. For strided registers, the first register takes the base value, and every following register adds the stride value to the former one.

TB-Level Compression

TB-level compression leverages inter-warp locality for vector registers. For example, in Figure 7, all threads across a TB has the same value of i when they are in the same dynamic program point. Such registers are defined as global uniform. A local uniform register is the registers that have the same value for PAG threads, e.g. variable ty. Similarly, global strided registers are the registers that are strided across all threads in a TB. For example, the index of s_weight, which equals to

$ty \times 16 + tx$, is global strided. Local strided registers are the registers that are strided for PAG threads but are not global strided, e.g. variable tx.

For local uniform/strided registers, the compression is the same as the warp-level compression. For a global uniform register, only one scalar register will be saved for the whole TB. Only two scalar registers, base and stride, in a TB will be saved for a global strided register. So, for global uniform/strided registers, the compression ratio is higher than warp-level compression.

Figure 11 illustrates the TB-level compression logic. For each logic register, the physical registers of all warps in a TB are analyzed by the warp-level compressor. The registers will be identified as random if any physical vector register is random. Then the random registers bypass the compressor and spill to the global memory. If all the registers are not random, the base and stride are stored in the base or stride vector buffer. After all warps in the TB finished compressing, the pattern of base vector and stride vector are analyzed. The registers are global uniform if the base vector is uniform and the stride vector is zero-uniform, meaning the scalar registers are all zeros. The registers are local uniform if the stride vector is zero-uniform and the stride vector is not uniform. The registers are global strided if the base vector is strided, the stride vector is non-zero uniform, and the stride in the base vector is the same as the stride in the stride vector. Otherwise, the registers are local strided. In this analysis, the base vector and stride vector width equal to TB_size/PAG.The maximum TB size is 1024, the minimum PAG we support is 8, and the N in Figure 11 is 32 in our implementation.

The TB-level register pattern is shown in Figure 9. The approach we use to analysis TB-level register pattern is similar to warp-level register patterns except that TB-level is only enabled when the preemption point is a barrier. If the preemption point is not at a barrier, the liveness of different warps may be different and the register pattern becomes difficult to analysis. So only the benchmarks with barriers are shown in the result. In Figure 6, we apply TB-level compression at barrier preemption points and warp-level compression for other preemption points. From the result, we can see that TB-level compression can further reduce register context size by 36.1% on average. However, because TB-level compression may be worse on some benchmarks, e.g. PF, and it can only be applied on barriers, we choose not to use TB-level compression for our preemption design.

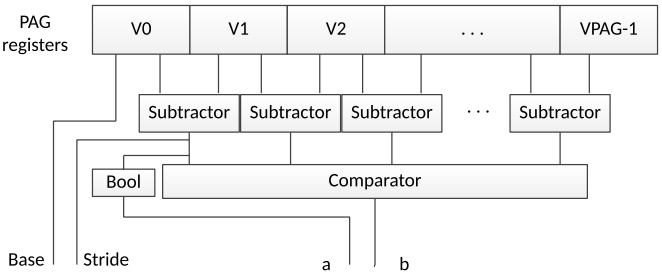

Fig. 10: Warp-level register state compression logic.

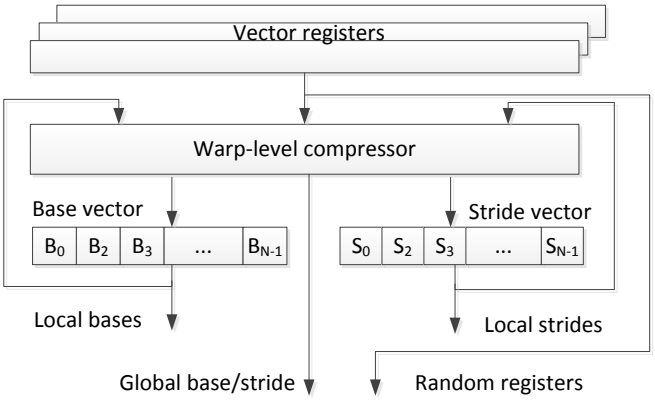

Fig. 11: TB-level register state compression logic.

IV. Context Switching for Preemption

A. Selective Preemption

As long execution time of a TB mainly results from loops with large numbers of iterations, we insert one preemption point for each loop. For nested loops, only the innermost loop is considered. For loops with one barrier, the barrier will be selected as the preemption point. If any other point is selected as a preemption point, deadlocks may occur when some warps are waiting at the barrier while other warps reach the preemption points and wait for preemption. The barrier with minimum liveness is selected by the compiler if there are more than one barrier in the loop. For loops without barriers, the point with minimum liveness is selected. Outside the loops, we insert one preemption point every K instructions. Similar to the loops, either the minimum liveness point or the barrier is selected. If a kernel does not have a loop or a barrier and the kernel is smaller than K instructions, the execution time of a TB is small and our approach is essentially the same as SM-draining [27]. In our experiment, because the execution time of all benchmarks is dominated by loops, the value of K does not have a great impact on the evaluation results when it varies from 100 to 1000.

We introduce two preemption point (pp) instructions, bar.pp and pp, to annotate the preemption points. After analyzing the preemption points, one preemption instruction is inserted to one point. For preemption point at barriers, bar.pp instruction is inserted to replace the original barrier instruction. bar.pp is a barrier instruction when the preemption signal is off. For the other preemption points, pp instructions are inserted into the program. The pp instruction becomes a nop when the preemption signal is off. When a preemption signal is on, warps keep running until a preemption point is reached. Then the warp stops and waits for preemption.

Both preemption instructions have one operand to provide the liveness bit vector for the program points at which the instructions are inserted. To follow the Fermi ISA format [1] [19], 10 bits are reserved as opcode. The remaining 54 bits are used as liveness bit vector. Because the architecture

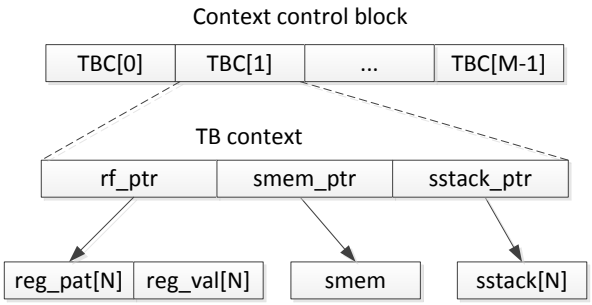

Fig. 12: Kernel context format.

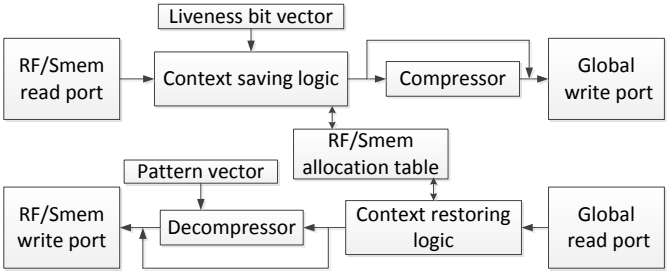

Fig. 13: Saving and restoring pipelines for preemption.

may support more than 54 registers, the highest bit is used to denote there are live registers that have higher register number than 53. All higher registers will be saved if such bit is set. In out benchmarks, we observe that a 53-bit vector is typically enough for representing the liveness. To provide the liveness for thread divergence, a dummy instruction is introduced and it follows the pp instruction. It also encodes 54 bits for the liveness bit vector. At runtime, if thread divergence is detected at the preemption point, the liveness which is encoded in the dummy instruction is used for preemption. The bar.pp instruction doesn't need to be followed by the dummy instruction because the barrier ensures that there should be no divergence [20].

B. Context Format

Due to the in-place context switching as we discussed in Section III-A, register file and shared memory can either be reserved on SM or dumped to global memory. In this paper, the context switching granularity is TB, meaning that the register file or shared memory of one TB cannot be partly spilled. But shared memory and that register file of one TB can reside in different locations, one in SM and the other in global memory, as illustrated in Figure 4.

As shown in Figure 12 is the context format of a kernel. The context control block (CCB) contains an array of TB context. The array size M is the maximum number of TBs that can be launched on the processor. Each entry contains the global memory pointers for the context of the register file, shared memory and SIMT stack. The pointer is NULL if the register file or shared memory is in-place reserved. Otherwise, a global memory space is allocated. The shared memory size on global memory is the same as the occupied size on GPU. The register file context on global memory has N entries, N equals to the warp number in one TB. The register context size for each warp is the maximum liveness number times vector register width. To maximize the bandwidth usage, the compressed register values are stored continuously. For decompressing, a pattern vector is used to store the register pattern and liveness of a warp. Two bits are used to represent the four states of each register. The four states are uniform, strided, random and dead. Because the maximum register number is 64, so the pattern vector length is 128 bits. In our paper, the whole SIMT stack will be saved to global memory. Because the SIMT stack includes the PC for each thread, the PCs are not separately saved. The warp is waiting at a barrier if the PC points to a barrier instruction, so the barrier state of each TB can also be derived from the SIMT stack.

C. Preemption Pipeline

Because the executions on different SMs are independent, a new kernel may preempt all SMs or only some of them. Here, we focus on preemption in one SM. When an SM receives an interrupt signal for preemption, the active warps keep executing until a preemption point is reached. Then the reached warp is set as inactive so that they will not fetch or issue new instructions. In order to preserve precise states, a warp must be drained before being switched out. A drained warp means that it has no issued instructions in the pipeline and has no pending updates to the register file.

As shown in Figure 13 is the spilling and restoring pipeline for preemption. For saving the context, the context saving logic looks up the register and shared memory allocation table, shown in Figure 4, to calculates how much resources (i.e. registers and/or shared memory) to be spilled to global memory in order to accommodate the new kernel. Such information is converted to how many resident TBs to be spilled. To save the register of a warp, the liveness vector is fetched from the instruction buffer. Then the live vector registers are compressed and pushed into a buffer. Because the most efficient way to access global memory is by a width of 128B, the compressed data form data segments with the size of 128B through the buffer. After the register states of all warps from one TB are drained, shared memory used by this TB starts to be spilled to global memory.

To restore a TB, the restoring logic waits for there is enough on-chip resource to launch the TB. Then the context control block, shown in Figure 12, is accessed to find the TB context. To restore the registers of a warp, the pattern vector is firstly loaded. Then each vector register is decompressed based on its pattern.

V. Experiments

A. Methodology

We implemented our lightweight context switching on GPGPU-sim [4] v3.2.2. Our baseline architecture models the NVIDIA GTX480 GPU, and its configuration is shown in Table I. GPGPU-sim supports both the PTX and GT200

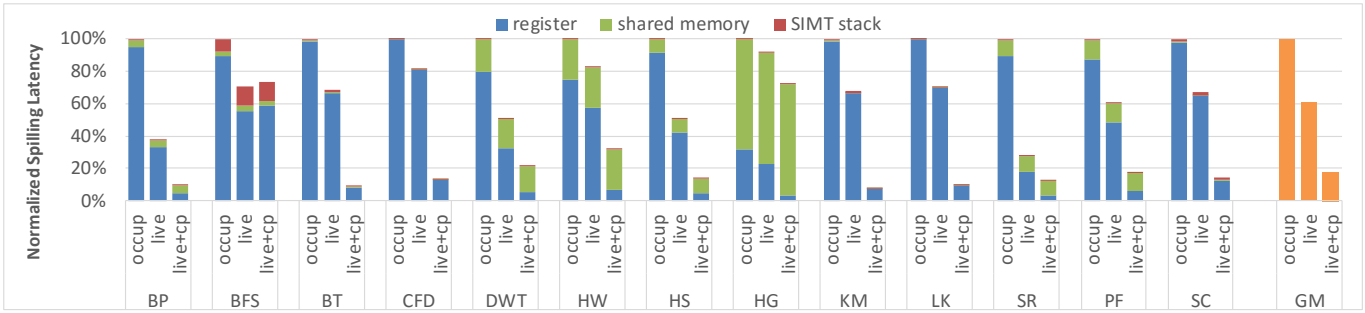

Fig. 14: Normalized spilling latency.

TABLE I: Baseline architecture configuration

Num. of SMs	15
SIMT core freq.	700MHz
Warp size	32
SIMD width	32
Resources per SM	8 TB slots, 48 warp slots (1536 threads), 128KB register file, 48KB shared memory
Warp scheduler	2 schedulers, RR policy
L1 D-cache	16KB per SM, 128B block size
L2 cache	128KB per channel, 6 channels
DRAM	924MHz, QDR, 384-bit bus, peak bandwidth = 0.924*4*384/8 = 177GB/s

instruction set architecture (ISA). PTX is for a virtual machine with unlimited registers. Therefore, in order to collect the accurate architectural register information, all benchmarks are compiled to the GT200 ISA. We evaluate our techniques on the Rodinia [6] benchmarks. Table II lists all the benchmarks. Each entry in Table II shows the information of a kernel. Because some benchmarks (e.g., BP) contain multiple kernels, (e.g., BP_1 and BP_2), we combine the results of these kernels in our evaluation.

We model the potential traffic contention due to context switching at register read/write ports, shared memory read-/write ports, the interconnect to memory controllers, and memory read/write bandwidth. For preemption, we found that the contention is limited as the SM essentially stops execution and all the ports are used for context swapping. The context switching requests have lower priority than regular requests from instruction execution.

To evaluate the preemption performance, we add periodic preemption signals (every 10000 cycles) when running the benchmarks. We run each benchmark for at most 200 million cycles or until it exits. When a preemption signal is received, one SM stops running and spill its architectural states to global memory while other SMs keeps running, the same method as used in prior works [22].

B. Spilling Latency

In Figure 14, we evaluate the normalized latency for spilling the architectural states to global memory. Three approaches are compared to show the effectiveness of liveness analysis and register compression. 'Occup' shows the latency to spill all the occupied architectural states. The spilling latency is measured from interrupt signal is issued to all the states are spilled. 'Live' is to spill only the live registers and 'live+cp' is to spill the live registers with warp-level register compression. For 'live' and 'live+cp', the latency for the spilling registers of a warp starts from the preemption point is reached. Then the latencies of all warps are accumulated. For each mechanism, the normalized latency to spill register file, shared memory and SIMT stack are evaluated.

If we see the results of 'Occup', the spilling latency of most benchmarks (except for HG) is dominated by spilling the registers. This is because the register file is the largest on-chip memory that stores the architectural states and it has relatively high occupancy (Figure 3). For DWT, HW and HG, shared memory accounts for more than 20% of spilling latency because the shared occupancy for these benchmarks is high (Figure 3). For most benchmarks, the SIMT stack spilling latency is too small to be observed. The SIMT stack latency appears for BFS because its live register number is very small.

From the results, we can see that liveness analysis and register compression drastically reduce the latency to dump the register state. Since shared memory size is not reduced by these two mechanisms, the latency for saving shared memory states is similar for different approaches. The geometric mean for spilling total architectural states is reduced to 17.7%. With the GPU core frequency of 700Hz, the average spilling latency is reduced from 9.9us to 1.8us. In the special case of BFS, the spilling latency is increased with 'live+cp' compared to 'live'. The reason is that no live registers can be compressed at the preemption points and our preemption mechanism needs to store metadata for the compression patterns of the registers. Therefore, the data to be saved become larger for BFS when compression is enabled.

C. Preemption Latency

The preemption latency evaluation is shown in Figure 15. The results are normalized with spilling all occupied architectural states to global memory. 'Select' shows the selective preemption latency with register liveness analysis and compression. The total preemption latency is measured from the

TABLE II: Benchmark specification

Kernel (Label)	Benchmark (Label)	Warps/TB	TBs/SM	Vec_reg/Warp	Smem/TB (bytes)	Limiting Factor
layerforward (BP_1)	backprop (BP)	8	6	13	1128	warp
adjust_weight (BP_2)	backprop (BP)	8	6	18	40	warp, reg
Kernel1 (BFS_1)	bfs (BFS)	16	3	7	44	warp
Kernel2 (BFS_2)	bfs (BFS)	16	3	4	36	warp
findRangeK (BT_1)	b+tree (BT)	8	6	10	48	warp
findK (BT_2)	b+tree (BT)	8	6	9	60	warp
initialize_variable (CFD_1)	cfd (CFD)	6	8	6	32	warp
compute_step_factor (CFD_2)	cfd (CFD)	6	8	8	48	warp, TB
compute_flux (CFD_3)	cfd (CFD)	6	4	39	36	reg
copySrcToComponets (DWT_1)	dwt2d (DWT)	8	6	4	804	warp
fdwt53Kernel (DWT_2)	dwt2d (DWT)	6	5	32	8668	reg, smem
kernel (HW_1)	heartwall (HW)	8	4	23	11888	smem
calculate_temp (HS_1)	hotspot (HS)	8	4	31	3144	reg
histogram1024 (HG_1)	hybridsort (HG)	3	3	10	12324	smem
invert_mapping (KM_1)	kmeans (KM)	8	6	9	32	warp
GICOV_kernel (LK_1)	leukocyte (LK)	6	8	18	24	warp, TB
lud_diagonal (LUD_1)	lud (LUD)	1	8	8	2076	TB
lud_perimeter (LUD_2)	lud (LUD)	1	8	16	3104	TB
lud_internal (LUD_3)	lud (LUD)	8	6	9	1056	warp
reduce (SR_1)	srad_v1 (SR)	16	3	14	4132	warp
srad (SR_2)	srad_v1 (SR)	16	3	16	128	warp
dynproc_kernel (PF_1)	pathfinder (PF)	8	6	12	2096	warp
kernel_compute_cost (SC_1)	streamcluster (SC)	16	3	8	56	warp

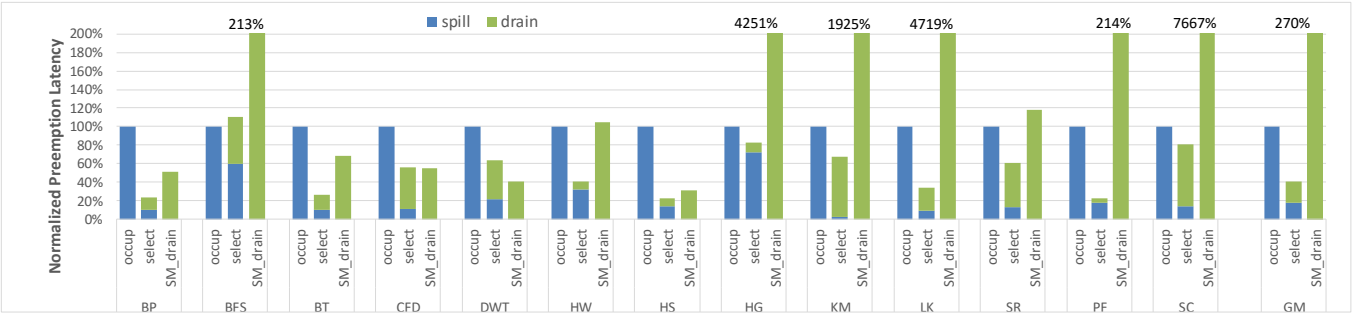

Fig. 15: Normalized preemption latency.

start of preemption signal to the architectural states are spilled. The latency labeled as 'spill' is the context spilling latency. With selective preemption, the warps keep executing until preemption points are reached. So, for some time the spilling pipeline is idle to wait for warps reaching the preemption points. Such latency is called draining latency and labeled as 'drain'. Because SM-draining lets all current TBs to finish, it doesn't need to save any architectural states for these TBs.

Compared with the baseline, the preemption latency is reduced to 40.3% on average (geometric mean). With the GPU core frequency of 700Hz, the preemption latency is reduced from 9.9us to 4.0us. The draining latency accounts for 55.8% of the selective preemption latency. Note that during draining, some (if not all) warps are still doing useful work.

For SM-draining, although the SM keeps doing useful work during preemption, the latency becomes unbearable for many benchmarks. For example, the average preemption latency for LK is 1431.1us. The newly incoming kernel would have to wait for such long TBs. As a result, fairness cannot be guaranteed with SM-draining because it favors kernels with long TBs. With selective preemption, because the preemption is guaranteed to be done in every loop iteration or every 1000 instructions, the draining latency is much more manageable.

D. Worst Case Preemption Latency

Because selective preemption has to wait for the warps to execute some instructions before being spilled, the latency variation may become higher than naive approach. To evaluate the preemption latency in the worst case scenario, we select 12 kernels which can run long enough to generate 15 times preemption signals. As shown in Figure 16, for each mechanism, the worst case preemption latency is normalized to its average latency. From the results, we can see that the naive approach, which is saving all occupied states, has 0.4x difference between average and worst case scenario. The difference may result from the different instructions to drain

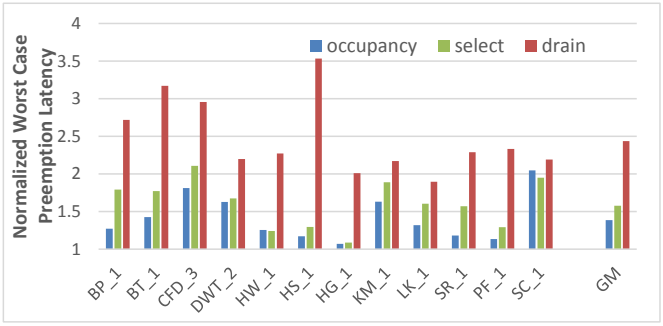

Fig. 16: Normalized worst case preemption latency

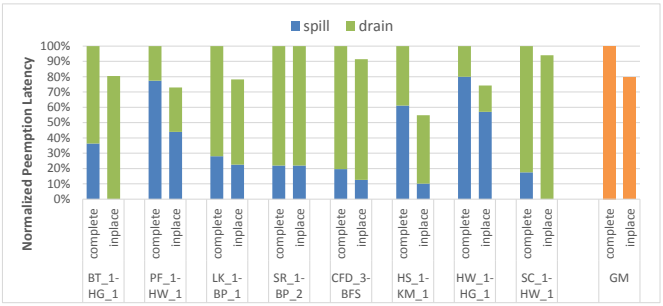

Fig. 17: Normalized preemption latency with in-place context switching

before preemption or the difference of memory traffic. For selective preemption, the difference between average and worst case scenario is 0.6x, which is slightly higher than the naive approach. The worst case latency for SM-draining is 2.4x compared with the average. For SM-draining, the worst case happens when an interrupt is signaled when a new TB has just being launched.

E. Impact of In-Place Context Switching

To evaluate in-place context switching, we randomly pick 8 pairs of kernels which are labeled as 'kernel1-kernel2' in Figure 17. We assume that 'kernel1' is preempted by 'kernel2' and measure the switching out latency for 'kernel1'. The baseline, which is labeled as 'complete', is the preemption latency of the approach using both liveness analysis and compression. For in-place context switched warps, they still have to reach the preemption point until being handled by preemption pipeline. As a result, the warps still need to be drained even if there is no register to spill, e.g. BT_1-HG_1. From the figure, we can see that the latency for spilling the register and shared memory states can be further reduced with our proposed in-place context switching, by 21.5% on average. On some benchmarks, e.g. BT_1-HG, The draining latency is higher on in-place context switching. This is because spilling can hide the latency for some warps to drain.

VI. Related Work

On CPUs, there are many works focusing on context reduction to reduce the preemption overhead and improve processor utilization. Some works [25] [30] propose to seek program points with small numbers of live registers for context switching, thereby reducing the context switching latency. Register relocation [28] is used to partition the register file into variable-size contexts. The more-often resident contexts are allowed to stay on the processor. Switching between resident contexts is very fast, and multiple contexts can tolerate long latencies from cache misses.

To enable fast context switching and exception handling on GPUs, iGPU [18] partitions kernel code into idempotent regions and each region is a recovery point. iGPU also leverages liveness analysis when formatting recovery points for context reduction. Register liveness is also used for dynamic register file management [13]. Lee et al. [17] leverage register compression for reducing GPU power. They use the base-delta-immediate (BDI) compression algorithm [23] for register file compression. BDI separates a vector register into several trunks and stores the value of first chunk and the delta between adjacent chunks. As delta values tend to be very small, they can be stored in small bins. The compression technique is used for the register file and every each register read/write needs to be decompressed/compressed. In comparison, we only perform compression/decompression when a context is spilled/restored.

Some recent works aim to enable the preemption on GPU. RGEM [14] is a user-space solution to reduce the response time of high priority kernels. It splits the input data into multiple chunks so that a kernel can be preempted at a chunk boundary. PKM [5] partitions the overall TBs of a kernel into multiple sets where each set has a specific number of TBs. Softshell [26] is a GPU programming model which supports a kernel being preempted at the boundary of TBs. In comparison, our proposed approaches enable efficient preemption at the instruction granularity.

Concurrent kernel execution is another option to support GPU sharing by multiple kernels. KernelMerge [11] and Spacial Multiplexing [3] study how to use concurrent kernels to better utilize GPU resources and improve overall throughput. Elastic kernel [21] increases GPU utilization by issuing concurrent kernels on one SM. After TBs from one kernel are issued to the SM, the spare resources are distributed to another kernel. In [16], Lee et al. also leverage mixed concurrent kernels to improve GPU utilization.

Compared to these prior works, the novelty of our work includes (a) fast context switching through context reduction and compression (b) efficient instruction-level GPU preemption.

VII. Conclusions and Future Work

In this paper, we present lightweight context switching for SIMT processors and compiler-hardware co-design to enable efficient preemption. We propose three schemes, in-place context switching, liveness analysis and register compression, to address the problem of the large kernel context on SIMT processors. Our results show that with register liveness analysis and compression, the register context can be reduced drastically by 91.5%. With selective preemption enabling instructions, we can achieve efficient instruction-level

preemption with an average preemption latency of 4.0us (with the 700MHz GPU core frequency).

Our work mainly focuses on the register context. For shared memory, Yang et al. [29] observe that the lifetimes of the shared memory variables are short and they propose explicit allocation and de-allocation functions. We can use the shared memory de-allocation points as the potential places for selective preemption. It provides finer granularity than TB-level context switching and reduces the shared memory context size. Integrating such shared memory management with our proposed schemes is left as our future work.

VIII. Acknowledgements

We would like to thank the anonymous reviewers for their valuable comments. This work is supported by NSF grants CCF-1216569, CCF-1618509, a Chinese research program "introducing talents of discipline to universities B13043", and an AMD gift fund.

References

[1] asfermi: An assembler for the nvidia fermi instruction set. [online]. available: https://github.com/hyqneuron/asfermi.

[2] M. Abdel-Majeed and M. Annavaram. Warped register file: A power efficient register file for gpgpus. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pages 412–423, Feb 2013.

[3] J. Adriaens, K. Compton, N. S. Kim, and M. Schulte. The case for gpgpu spatial multitasking. In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, pages 1–12, Feb 2012.

[4] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 163–174, April 2009.

[5] C. Basaran and K.-D. Kang. Supporting preemptive task executions and memory copies in gpgpus. In *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*, pages 287–296, July 2012.

[6] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, 2009.

[7] S. Collange, D. Defour, and Y. Zhang. Dynamic detection of uniform and affine vectors in gpgpu computations. In *Proceedings of the 2009 International Conference on Parallel Processing*, Euro-Par'09, pages 46–55, Berlin, Heidelberg, 2010. Springer-Verlag.

[8] M. de Kruijf and K. Sankaralingam. Idempotent processor architecture. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 140–151, New York, NY, USA, 2011. ACM.

[9] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt. Dynamic warp formation: Efficient mimd control flow on simd graphics hardware. *ACM Trans. Archit. Code Optim.*, 6(2):7:1–7:37, July 2009.

[10] M. Gebhart, S. W. Keckler, B. Khailany, R. Krashinsky, and W. J. Dally. Unifying primary cache, scratch, and register file memories in a throughput processor. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 96–106, Washington, DC, USA, 2012. IEEE Computer Society.

[11] C. Gregg, J. Dorn, K. Hazelwood, and K. Skadron. Fine-grained resource sharing for concurrent gpgpu kernels. In *Presented as part of the 4th USENIX Workshop on Hot Topics in Parallelism*, Berkeley, CA, 2012. USENIX.

[12] K. Gupta, J. Stuart, and J. Owens. A study of persistent threads style gpu programming for gpgpu workloads. In *Innovative Parallel Computing (InPar), 2012*, pages 1–14, May 2012.

[13] H. Jeon, G. S. Ravi, N. S. Kim, and M. Annavaram. Gpu register file virtualization. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 420–432, New York, NY, USA, 2015. ACM.

[14] S. Kato, K. Lakshmanan, A. Kumar, M. Kelkar, Y. Ishikawa, and R. Rajkumar. Rgem: A responsive gpgpu execution model for runtime engines. In *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, pages 57–66, Nov 2011.

[15] J. Y. Kim and C. Batten. Accelerating irregular algorithms on gpgpus using fine-grain hardware worklists. In *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pages 75–87, Dec 2014.

[16] M. Lee, S. Song, J. Moon, J. Kim, W. Seo, Y. Cho, and S. Ryu. Improving gpgpu resource utilization through alternative thread block scheduling. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 260–271, Feb 2014.

[17] S. Lee, K. Kim, G. Koo, H. Jeon, W. W. Ro, and M. Annavaram. Warped-compression: Enabling power efficient gpus through register compression. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 502–514, New York, NY, USA, 2015. ACM.

[18] J. Menon, M. De Kruijf, and K. Sankaralingam. igpu: Exception support and speculative execution on gpus. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 72–83, Washington, DC, USA, 2012. IEEE Computer Society.

[19] NVIDIA. cuda binary utilities. [online]. available: http://docs.nvidia.com/cuda/cuda-binary-utilities/index.html.

[20] NVIDIA. *CUDA C Programming Guide*. March 2015.

[21] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan. Improving gpgpu concurrency with elastic kernels. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 407–418, New York, NY, USA, 2013. ACM.

[22] J. J. K. Park, Y. Park, and S. Mahlke. Chimera: Collaborative preemption for multitasking on a shared gpu. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 593–606, New York, NY, USA, 2015. ACM.

[23] G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry. Base-delta-immediate compression: Practical data compression for on-chip caches. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 377–388, New York, NY, USA, 2012. ACM.

[24] M. Rhu and M. Erez. The dual-path execution model for efficient gpu control flow. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pages 591–602, Feb 2013.

[25] J. S. Snyder, D. B. Whalley, and T. P. Baker. Fast context switches: Compiler and architectural support for preemptive scheduling. *Microprocessors and Microsystems*, 19:35–42, 1995.

[26] M. Steinberger, B. Kainz, B. Kerbl, S. Hauswiesner, M. Kenzel, and D. Schmalstieg. Softshell: Dynamic scheduling on gpus. *ACM Trans. Graph.*, 31(6):161:1–161:11, Nov. 2012.

[27] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero. Enabling preemptive multiprogramming on gpus. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 193–204, June 2014.

[28] C. A. Waldspurger and W. E. Weihl. Register relocation: Flexible contexts for multithreading. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 120–130, New York, NY, USA, 1993. ACM.

[29] Y. Yang, P. Xiang, M. Mantor, N. Rubin, and H. Zhou. Shared memory multiplexing: A novel way to improve gpgpu throughput. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 283–292, New York, NY, USA, 2012. ACM.

[30] X. Zhou and P. Petrov. Rapid and low-cost context-switch through embedded processor customization for real-time and control applications. In *Design Automation Conference, 2006 43rd ACM/IEEE*, pages 352–357, 2006.