GPGPU Register File Management by Hardware Co-operated Register Reallocation

Hyeran Jeon and Murali Annavaram

Computer Engineering Technical Report Number CENG-2014-05

Ming Hsieh Department of Electrical Engineering – Systems
University of Southern California
Los Angeles, California 90089-2562

May 2014

GPGPU Register File Management by Hardware Co-operated Register Reallocation

Hyeran Jeon Murali Annavaram
University of Southern California
{hyeranje,annavara}@usc.edu

Abstract

To support massive parallel threads context, GPGPUs use a huge register file. Due to their size, register file is one of the most power hungry logic in GPGPU. However, the current trends indicate that GPGPU register file size will continue to get even bigger as the demand for higher single instruction multiple thread (SIMT) parallelism increases, particularly in high performance application domain. In order to reduce power consumption demand, in this work, we exploit a fundamental observation that fairly high portion of the register spaces are unnecessarily allocated and burn power due to the fact that the registers are considered as private resource for each warp. For example, even though a register's value is no longer used by any instruction, the register space should be occupied by the warp during the program execution. In GPGPU, as single register usage in a code context leads to thousands of register space allocation, the power and space overhead due to any wasted register is significant. Instead, we propose to share the register file across warps. In our proposed allocation, a register is released from its physical register space immediately after its last use. Then, the released register space is reassigned to another warp's register. The compile time register lifetime analysis information is used for providing hint to the hardware about each register's release point. To enable this register reassignment, we propose a light weight register renaming in hardware. By releasing registers and reusing them across warps, we can reduce the demand for register file size on average by 30% compared with the optimally compiled applications. The reduced live register space leads to an average of 23% and 26% static power saving over a basic sub-array level and individual register level power gating.

1. INTRODUCTION

GPGPUs provide massive register file to quickly switch between thousands of thread contexts. To run thousands of threads concurrently, GPGPU needs to save and restore the architecture state of the threads on each thread switch. Since GPGPUs potentially can switch between threads every cycle, they can ill-afford to save context to an off chip memory or even a cache on die. Instead, GPGPUs have a register file that is multiple times bigger than the register file on traditional CMPs. The trend in GPGPU design indicates that with technology progression more thread contexts will be supported in future designs. For instance, the size of the register file per each streaming multiprocessor (SM) doubled from Fermi to the Kepler architecture [17]. The bigger size register file is not only expensive but also consumes significant leakage power. In fact, a recent power breakdown of GPGPU microarchitectural blocks showed that register files consume 13.4% of the total chip power [14] and 50% of that power is spent in leakage [11]. Several recent studies address the power consumption concern of GPGPU register file [9, 6, 10]. Some of these studies focused on reducing dynamic power by adding small register cache or multi-level register file [9, 10] thereby minimizing the access to the large main register file. One recent study focused on saving leakage power by forcing the inactive registers into low power mode using drowsy voltage.

However, these studies rely on the compiler determined register allocation. Our experimental result shows that the compiler determined register usage is over-provisioned. The compile time allocated registers are not fully utilized during the program execution due to two main reasons: various register lifetime and warp scheduling time differences. In GPGPU, warps are scheduled in different point in time. Especially in the state-of-the-art two level scheduler [9], as a small set of warps are scheduled in a ready queue until they cannot proceed due to long latency memory stalls, the execution time window of the warps in the ready queue and those in the pending queue is several hundred or thousand cycles away. When there is a register that has fairly short lifetime such that the value is not referenced across the scheduling, the register space is left unused during the hundreds of cycles while the warp stays in the pending queue. If a power management is not used in the GPGPU, the register space burns significant leakage power without contributing program correctness.

Our primary intuition is to reuse such wasted register spaces. As the registers are considered as a private resource of each warp, the unused register spaces cannot be reused by another warps. To avoid any resource conflict, compiler allocates an exclusive copy of register sets for each warp even though not every warp is scheduled in the same execution time window. Instead, we propose to share the register space across the warps. The registers are allocated in run time by using the compiler generated register lifetime information. By proactively reallocating the wasted register spaces for the other warps' registers, the total register usage as well as the corresponding leakage power can be reduced.

In GPGPU kernels, simple compiler analysis can detect when each register's value is lastly consumed (a.k.a. dead register). The dead register's physical space can then be reassigned to hold the contents of the next register that just begins a new life. By enabling such physical register space sharing, we can significantly reduce the number of live registers thereby decreasing power consumption and the demand for the register file size. In the rest of this paper, we first present the motivational data showing register lifetimes in GPGPU kernels. We then present the necessary software and hardware support to enable physical register sharing across warps.

To summarize, the followings are the contributions of this paper:

(1) We show and analyze the underutilization in the compile time allocated register spaces during the execution. To our knowledge, this is the first paper that identifies the underutilization in the compiler reserved register space in SIMT processors.

(2) We extend the existing compile time techniques to identify when each register is dead in the code and when the dead register can be released by considering the warp level thread execution. Then, we mark that information as part of the GPGPU binary.

(3) We propose to reuse the physical space for the dead registers across different warps. When a register is dead in one warp, its physical space will be reassigned to any other warp that has a need for another register space. We propose a simple register renaming hardware that enables cross-warp register sharing. The compile time information encoded in the kernel binary is conveyed to the hardware to make an accurate decision on when to release a register and reassign to another warp.

(4) We evaluate the proposed idea and show that applications can run with average of 30% less register space even compared to the optimally compiled workloads while saving 23% and 26% of the register leakage power consumption compared to a sub-array level and an individual register level power gatings with only 1% area overhead to store the register renaming table.

2. BACKGROUND

2.1. GPGPU Register File

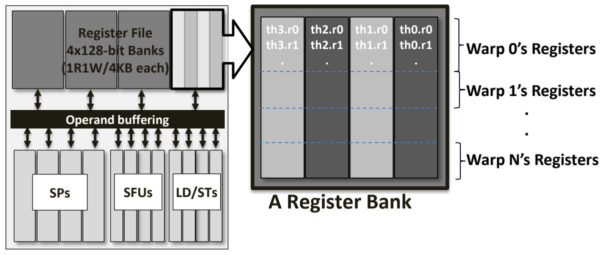


Figure 1: Baseline GPGPU register file

In this paper, we will use an NVIDIA Fermi GPGPU microarchitecture for evaluations. In this architecture, each chip has multiple streaming multiprocessors (SMs). Each SM consists of eight SIMT clusters that is comprising of four SIMT lanes, thereby allowing 32 SIMT lanes to be executed concurrently. Each SIMT cluster has 4 CUDA cores that can do a variety of integer, floating point, ld/st and special function operations. Each cluster accesses four register banks to find its input and output operands. Each entry of a register bank is 128-bit wide and contains four 32-bit registers, each associated to one SIMT lane within a cluster [9]. As each entry of the register bank consists of 4 registers having the same name but associated with 4 different threads, loading an entry from a register bank can feed all 4 SIMT lanes at once. Most common instructions that read 2 operands, write 1 result (*2R1W*), as well as the special instruction like MULADD that read 3 operands, write 1 result (*3R1W*) can access the four register banks to read their input operands and write output data concurrently without any register port stalls most of the time. However, if an instruction fetches operands from the same bank, the operands cannot be fetched concurrently. To handle bank conflicts, GPGPUs use operand collector buffering logic that hides the latency of multi-cycle register fetch. Each register bank is shared by multiple warps by allocating different register regions to each warp.

Due to the access latency as well as dynamic power, huge register files are typically partitioned into several small subarrays [12]. While there is no publicly available documentation on GPGPU register file organization, we believe the structure that is described in [12] is efficient for GPGPU register file. In such an organization, a 128KB register file in Fermi is partitioned into 32 4KB banks and then each 4KB bank uses four 1KB sub-arrays. When a data is to be fetched, one or more sub-arrays can be accessed. In this paper, we assume that an entire 128-bit register entry is loaded from one of the four sub-arrays.

2.2. Vendor's Effort for Power Efficiency

As the power efficiency becomes as one of the top priority design issues, vendors started to optimize the GPGPU architecture to reduce the power consumption. One major innovation that is made in NVIDIA Kepler is that the scoreboard logic becomes simpler. The researchers found that dependency tracking among the instructions can be removed from the scoreboarder by leveraging three factors: 1) instructions are scheduled in order fashion 2) the execution latency of the instructions are fixed in GPGPU, and 3) data dependency can be easily detected in compile time. Instead of tracking the data dependency in run time, Kepler uses a dependency information that is generated by compiler [17].

NVIDIA barely reveals the details of the design but a recent study found that a metadata instruction is added per seven instructions when a code is compiled for Kepler GPU [13]. They found that the format and the operation of the metadata instruction is similar to explicit-dependence lookahead that was originally used in Tera computer system [7]. For example, the information contained in the metadata instruction is used for indicating the cycles that the seven following instructions should wait until the dependencies are resolved. By using this information, the instructions can be scheduled in time without dynamically checking the dependencies. To pre-process the metadata instruction, the pipeline of Kepler is slightly changed from Fermi [17]. The fetch stage is partitioned into two separate stages: *Sched. info* and *Select*. Sched. info stage pre-processes the metadata instruction and Select stage selects an instruction to issue according to the metadata. The other pipeline stages are the same. In this paper, we leverage this new architecture to support our proposed design.

3. INEFFICIENCY IN COMPILE TIME REGISTER ALLOCATION IN SIMT PROCESSORS

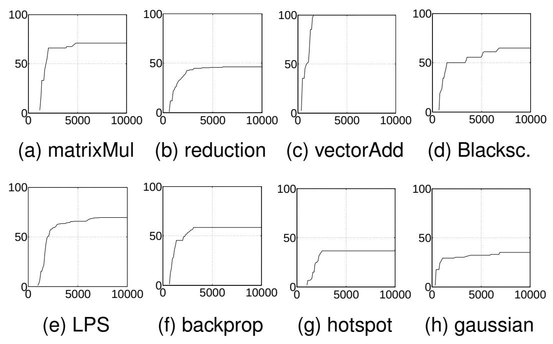


Figure 2: Utilization of the registers that are reserved by compiler in the first 10K cycles of execution (X-axis: cycle, Y-axis: utilization(%))

To support highly parallelized applications, vendors over-provision the register file size in GPGPU. In some applications, the register file is not fully utilized due to the program's lacking parallelism. The underutilized register space can be easily detected by using compiler generated register usage information. For example, if a kernel code uses N registers and the kernel is executed by M concurrent cooperative thread array (CTA)s that uses K warps each, the total register usage for the kernel per SM is $N \times M \times K \times 32$ when a warp has 32 threads. If the total available registers in a SM is T, the underutilized register space can be easily calculated by $T - (N \times M \times K \times 32)$.

However, from an experiment, we found that the allocated registers themselves are not fully utilized during the execution. Figure 2 shows the portion of registers that are actually used *among the registers reserved by the compiler* in a 10K cycles of the execution. X-axis and Y-axis respectively denote time in cycle and register utilization in percentage. The utilization trend is captured from eight representative applications of CUDA SDK, rodinia and Parboil in their first 10K cycles. We only counted the registers that are storing a valid value that is referred by any of the following instructions. We excluded the registers whose value lifetime is ended from the utilization calculation. Except vectorAdd that reaches 100% utilization in around 2000 cycle, the rest seven applications barely use more than half of the allocated registers during the monitored time frame.

This kind of underutilization is caused mainly due to the various register lifetimes and the nature of SIMT execution. A GPGPU application is executed by multiple warps. Warps are scheduled in different point in time. In some state-of-the-art scheduler that schedules the warps in two levels [9], to effectively hide the latency of long memory operation, the scheduling time difference between the warps that are in the ready queue and the pending queue can reach thousands cycles depending on the application. If there is a register that has its value lifetime ended before the warp is scheduled out from the ready queue to the pending queue, and then the new value lifetime begins when the warp is scheduled back in later, the corresponding register space is remained unused during the period that the warp stays in the pending queue. The period can last hundreds to thousands of cycles. Given that a warp has 32 threads and each thread uses its own register, the number of inactive registers can easily become several hundred.

As the compiler does not know such warp scheduling information apriori, the compilers should allocate an exclusive register storage for each warp by assuming that all the assigned warps are concurrently executed and the corresponding registers are perfectly utilized. However, such over-provisioned register allocation leads to unnecessary power consumption as the registers burn power once they are allocated even though some of them are not used any longer. Also, due to the exclusive register allocation per warp, the parallelism and performance that can be improved by using more registers can be limited. In this paper, we focus on improving power efficiency by exploiting the inactive register storages. The detailed register usage pattern in GPGPU and the novel power efficient register allocation method are explained shortly.

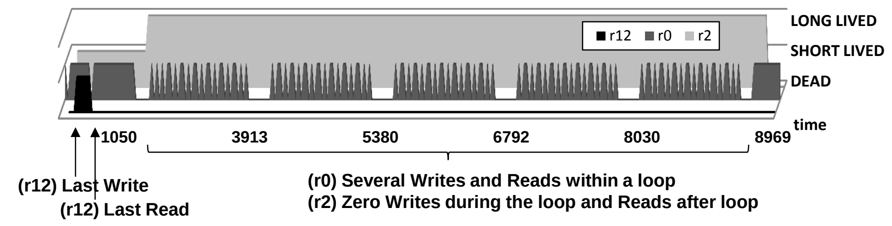
4. LIFE-TIME AWARE REGISTER ALLOCATION

4.1. Register Usage Patterns in GPGPU

Figure 3(a) shows three representative register usage patterns seen in GPGPU applications. The pattern is actually taken from the benchmark *matrixMul* of CUDA SDK used in our experimental evaluation, which is compiled with default options. We captured the lifetime of three registers, namely $r0$, $r2$ and $r12$. The X-axis represents time and Y-axis represents three different lifetime scenarios. A dead value is represented with a Y value of zero, a short lived register is represented as a next step up in the Y-axis and the last step up represent the long lived registers.

Dead register contains the value that is no longer consumed until next write to the register. In this figure, $r12$ has the shortest lifetime. It is only used for a short time scale of just a few cycles at the beginning of the program and it is then dead. In particular, $r12$ lifetime is restricted to the start of the program execution but before the beginning of a loop. Since the compiler is aware that this register is dead at the end of last read, it tries to reassign the register for a different compu-

(a)



(b)

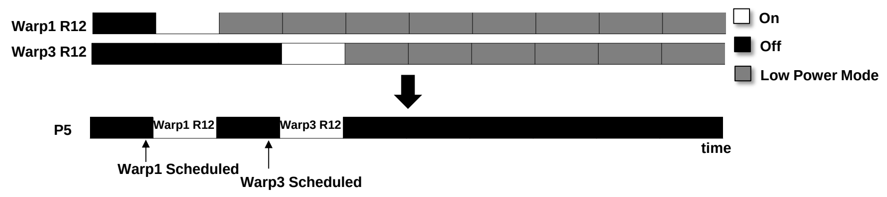


Figure 3: (a) Register lifetime variance (CUDA SDK matrixmul example) and (b) the expected power efficiency by using register renaming: warp1's r12 is mapped to p5 only when it is alive and then released to reuse p5 for mapping to warp3's r12. p5 is released and can be completely turned off after warp3's r12 is dead.

tational purpose after its last use. However, in this code, the loop body uses only few registers including $r0$ and then the kernel terminates shortly after the loop iterations. Even in a single threaded application, wasting a single register is an inefficient use of space. But in the GPGPU context, all the threads' $r12$ registers have the same lifetime pattern. For instance, the $matrixMul$ kernel uses 40 warps, for a total of 1280 threads per SM. Hence, 1280 values of register $r12$ are dead for very long time. These dead registers burn leakage power without contributing anything to the program correctness. The accumulated register space for $r12$ across all threads is 5KB which is bigger than a register bank. As current GPGPU register file management does not consider a register's lifetime, $r12$ cannot be powered off.

On the other hand, $r2$ is written at the beginning of the program and is read at the end of the program execution. As $r2$'s value is read at the end of the program, the register is alive for the entire program duration. Clearly $r2$ is a long lived register.

The last register in this example, $r0$, is actively produced and consumed within a loop. Each group of spikes is a loop iteration and there are a total of five loop iterations shown in the figure. On each entry into the loop, $r0$'s value is loaded from global memory and then a series of read-write sequences are performed. In each iteration, $r0$ stores a value which is then consumed within a few cycles and then a new value is written immediately after last read. At the start of a new iteration, a new value is loaded from memory and the process repeats. This is an example of a register that has multiple lifetimes but each lifetime is relatively short. At the beginning of each iteration, there is a long latency in accessing global memory (typically, 400-600 cycles) to load the value of $r0$. However, using the two level warp scheduler [9], after each iteration, when a warp encounters a long latency operation such as ld, the scheduler schedules the warp out from the ready queue to the pending queue. Only when the data arrives from the memory, the warp is scheduled back in to the ready queue. Because of such scheduling policy, in between two iterations there is no need to hold on to the physical register space assigned to $r0$.

4.2. Opportunistic Register Reuse

To effectively reduce the wasted register space and corresponding power dissipation, we propose to share the register space across the warps. As explained in the previous section, warps are scheduled to execute the same code at different points in time. When a register life time ends in a warp, that register can be allocated to a different warp which is beginning a new register life cycle. If the register value lifetime is known apriori, a short lived register can then be released by one warp, and the space that was allocated to the register can be reused by warps that are scheduled later.

Figure 3(b) shows an example of register reuse. Warp one and three execute the same code but scheduled in different point in time. Therefore, a short lived register $r12$ is used by warp one and warp three in different time slots. Recall that from Figure 3(a) $r12$ is only needed in each warp before the loop execution. If warp one releases $r12$ right after its valid lifetime that is illustrated as white rectangle, warp three can reuse the space for its own $r12$ storage. In addition, a register that is waiting for a long latency operation, such as $r0$ in between two loop iterations, can also be reused during the long stall time. Furthermore, if there are any two CTAs that do not have their execution times overlapped (for example, the trailing CTA begins the execution after the leading CTA almost finishes its execution), all the registers including the long lived registers such as $r2$ that are assigned to the leading CTA can be entirely reused by the trailing CTA. Thus the number of registers used in the system can be reduced.

Then, we can also reduce the power consumption by only activating the decremented number of active registers and turning off (i.e. power gating) the inactive registers. The details on the power gating is described in Section 4.4.

To enable register sharing across warps, it is necessary to separate architectural registers from the physical register space they occupy. Conventional CPUs have long used register renaming to avoid false data dependency by mapping an architectural register's multiple value instances to distinct physical registers. We propose to adopt similar technique but in a completely different way and purpose within the GPGPU execution environment. The renaming table in our proposed approach is used for mapping multiple architectural registers to a physical register to reduce the unused register file space as well as power consumption. Also, a register renaming logic is designed to use the register lifetime information that is provided by the compiler, while the CPU side renaming is operated purely by hardware. In the next section, we describe how register lifetime information can be gathered by the compiler.

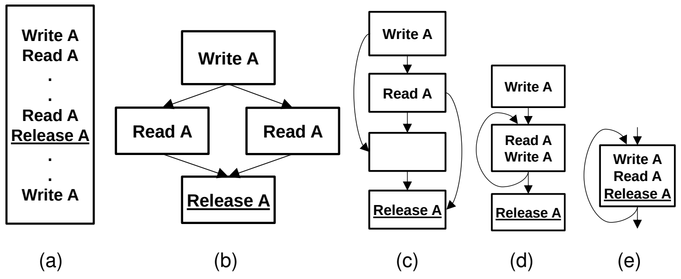


Figure 4: Register Release time w.r.t. Lifetime Analysis

4.3. Register Lifetime Analysis

Intra-Basic Block: In order for the register management logic to release a register, it has to track register lifetime. Rather than dynamically tracking the lifetime, we will rely on compiler to statically identify the life cycle begin and life cycle end points in the code and pass this compile time information to the hardware. Figure 4 shows five representative code examples that should be considered by the compiler in register lifetime analysis. Each rectangle represents a basic block. In the first scenario shown in Figure 4(a) an intra-basic block analysis can be done trivially to determine lifetime. Whenever a register is used as a destination operand of an instruction, the previous instruction that uses the register as a source operand can release the register after reading the value. We add one meta data bit flag per each operand for each instruction to indicate each source operand's release time. As CUDA instructions have maximum of three operands, three bits are used per instruction and these metadata bits are called per-instruction release flag (*pir*). When a bit is 1, the corresponding operand storage register can be released after it is read by the current instruction. More details about these metadata bits and their organization are described shortly.

Diverged flows: In the presence of a branch divergence, the register release information must be conservatively set because registers are released in warp level. For example, if any instruction in a diverged flow mistakenly release a register, the value for the threads that should execute another diverged flow will be lost. Figure 4(b) and (c) show two scenarios. The register is defined before entering the basic block and it is used within the two diverged flows as shown in (b). In this case, the register can be safely released only at the reconvergence point. The reconvergence point can easily be computed using standard compile time immediate post dominator basic block identification. However, the main problem is that unlike in the intra-basic block case, here the register release is not associated with the actual last use instruction of the register. Instead, it is associated with an instruction that happens to start at the reconvergence point. It is also possible that multiple registers may need to be released at the reconvergence point. Hence, rather than adding meta data to an existing instruction, we add a new per-branch release flag (*pbr*). The flag contains the list of architectural register IDs that can be released at the start of the reconvergence block.

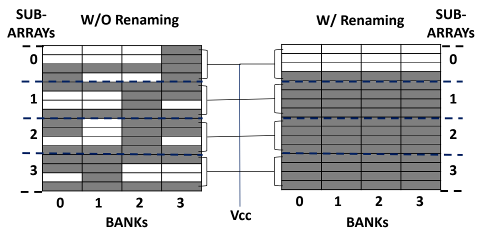
Figure 4(c) shows a more complex nested diverged flow. But the essential principle for when the register release can occur is based again on the immediate post dominator analysis, which is routinely carried out in many compiler optimization passes.

Loop: Figure 4(d) shows a loop where a register produced in one iteration is used in another iteration. In this scenario, clearly there is no option to release the register until all iterations are complete. If on the other hand, there is no loop carried dependence on registers across loop iterations, then it is possible to release the register after the last consumption within the loop body as shown in Figure 4(e). Therefore, when a register value is referenced across the iterations or outside of the loop, it can only be released outside of the loop.

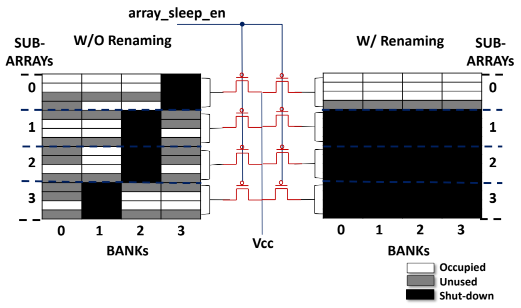
4.4. An Usecase: Power Gating

In this section, we are going to show how the proposed register allocation can help for power efficiency. The register lifetime analysis allows us to turn off all the dead registers. Without the information, the dead registers burn leakage power without contributing any program correctness. We explored two different types of power gatings in the evaluation: 1) individual register level power gating and 2) sub-array level power gating. However, in this section, we only show the conceptual illusion that the register renaming can bring to the power efficiency. The details are described in Section 8.3.

Figure 5 shows an example when sub-array level power gating is used. Each of the two rectangles in (a) is the register file in a SIMT cluster. The four columns denote the four register banks and each entry contains a 128-bit register. The white entries are the active registers and the gray ones are unused register entries. The four horizontal partitions separated by dotted lines show the four sub-arrays. For simplicity, a common power line is plugged to the two SIMT clusters' register files in the figure but the two register files are independent. The left hand side register file shows the active registers distribution when the typical register allocation is used and the right hand



(a) W/O Power Gating



(b) W/ Power Gating

Figure 5: Effect of register renaming: when register renaming is used, (a) active registers are consolidated into a sub-array in all four register banks (b) more sub-arrays can be turned off as registers are consolidated in less number of sub-arrays (power line and sleep transistors are simplified just for the illustration)

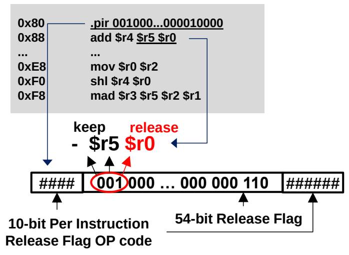
side register file shows the one when the proposed register renaming is used. By using the register lifetime information given by the compiler, the number of active registers can be firstly reduced. Then, by using the architectural register to physical register mapping, the active registers are consolidated into one sub-array in each bank.

In (b), a sleep transistor is added to each sub-array for sub-array level power gating. The figure is again, simplified for the explanation so, the sleep transistors and the Vcc of the two register files are not connected in the real design. As shown in the right hand side register file, the sub-arrays from 1 to 3 in all four register banks can be turned off as there is no active registers in those sub-arrays when register renaming is used. On the other hand, without register renaming, as the registers are scattered across the sub-arrays, each register bank can turn off only few sub-arrays; all four sub-arrays in register bank 0 are active. If all the active and unused registers equally burn leakage power, the register file without register renaming burns 8 times higher leakage power than the register file that uses the register renaming.

5. COMPILER SUPPORT

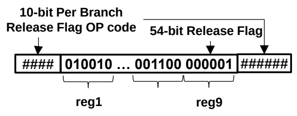
5.1. Per-instruction Release Flag Generation

The register lifetime information is generated at compile time and embedded in the code. As mentioned earlier, each instruction has a three bit per-instruction release flag, where each bit indicates one of the maximum of three source operands



```
0x80  .pir 001000...000010000
0x88  add $r4 $r5 $r0
...   ...
0xE8  mov $r0 $r2
0xF0  shl $r4 $r0
0xF8  mad $r3 $r5 $r2 $r1
```

(a) Per instruction release flag



(b) Per branch release flag

Figure 6: Two release flag instructions

that can be released. If the bit is 1, the corresponding operand register can be released after read by the instruction. But embedding a 3-bit *pir* in each instruction requires significant modification on the instruction fetch and cache access logic. To avoid this concern, we add a 64-bit flag set instruction that is present at the beginning of each basic block as shown in Figure 6(a). The selection of 64-bit flag is to accommodate the fact that CUDA code is 64-bit aligned. The flag set instruction consists of a 10-bit register release opcode, and 18 three-bit flags that can cover 18 consecutive instructions within the basic block. If a basic block is larger than 18 instructions long, a flag set is inserted every 18 instructions. If the basic block has fewer than 18 instructions then some of the flag bits are simply unused. Note that the 10-bit register release opcode is split into two sets of four and six bits to follow the Fermi instruction encoding format, which wraps the opcode at the MSB and LSB bits of the instruction [1, 16].

In the example of Figure 6(a), the *pir's* first three bit is the release information for the first *add* instruction. Each of the three bit, 001, denotes the release point of corresponding operands. Let us assume that $r0 is determined to be dead at the *add* instruction according to the register lifetime analysis. Since *r*0 is the first input operand, the corresponding *pir* flag bit is set to one. The second *r*5 is still alive and hence the corresponding flag is set to zero. There is no third input operand for the *add* instruction and hence the corresponding bit in the *pir* is a don't care. The decoding process of the flag instruction is explained shortly.

5.2. Per-branch Release Flag Generation

At the diverged flows, we do a conservative release. The registers that are referenced across multiple flows or loop iterations are only released when the diverged flows are converged. At the reconvergence point, a *pbr* is added. As shown in Figure 6(b), the format is similar to *pir*. The only difference is that every six bits present a register number to release. Note that each thread in Fermi can use up to 63 registers which can be identified by six bits. Total of nine registers can be covered by a *pbr*. If more than nine registers are to be released, more pbrs are added. However, according to our evaluation, the average number of registers that are released by *pbr* is less than 2.

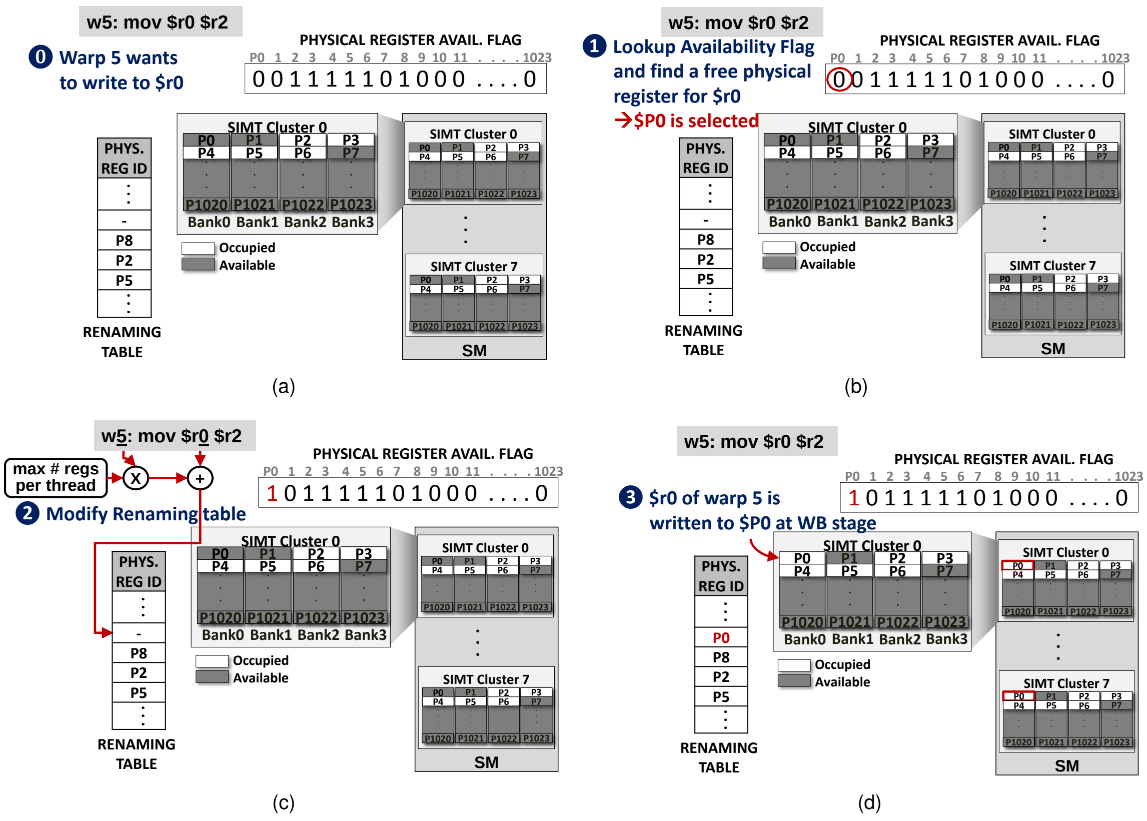


Figure 7: An example of the proposed register allocation: 1) physical register availability flag is looked up to find an available physical register, 2) renaming table is updated, and then 3) the reserved physical register is written

6. ARCHITECTURE SUPPORT

6.1. Renaming Table

To enable register reallocation, we use the concept of register renaming. Each architectural register is mapped to a physical register whenever it is written. When the architectural register value is no longer used, the mapping is released. The release point is provided either as a part of the *pir* or *pbr*. Once a physical register is released, the physical register is marked as available which can then be remapped to another architectural register in the future. The physical availability marking is explained shortly.

To maintain the mapping information, a register renaming table is added to each SM. Since registers are allocated and released per warp, the renaming table is operated per warp. Each renaming table is indexed by a combined id of warp id and architectural register id and contains the corresponding physical register id. The total renaming table size per SM is calculated like below:

$$\textit{Renaming Table Size} = \#\,\textit{warps per SM} \times \#\,\textit{regs per kernel} \times \log_2(\textit{max physical register count})\ \textit{bits} \tag{1}$$

Each SM has a 128KB register file in the Fermi architecture and each register width is 32 X 32-bit registers each providing data to one SIMT lane. Hence, there are a total of 1024 physical registers. Therefore, each entry of the renaming table is 10 bits long. In Fermi, that uses maximum of 48 warps and 63 registers per thread, the renaming table size is 3.69KB which is 2.8% among 128KB register file.

To find an available physical register when mapping a new register, a 1024-bit physical register availability flag is also used. Each bit indicates whether the corresponding physical register is occupied or not. To summary, the total area overhead for register renaming is 3.8KB which is 2.9% among 128KB register file. However, this overhead is significantly reduced with an optimization that will be described Section 7.

The renaming table consists of four banks so that the operands can lookup the physical register id concurrently. When there is a bank conflict, the name lookup can be serialized. The pipeline modification to access renaming table is illustrated in Figure 8. According to our simulation, the access latency of the optimized renaming table (will be described in the following section) is 0.22ns which is 17.6% of a typical cycle period (1.25ns) of GPGPU of 40nm technology [2]. Therefore, the renaming table lookup can be done during the operand collector stage that typically takes two to three cycles.

Figure 7 shows an example sequence of the proposed register allocation. A SM has a renaming table, eight register files (one per SIMT cluster), and a physical register availability flag. Each register file consists of four banks. Assume that *warp*5 tries to store a value to \$r0 and the source register \$r2 is not released at this instruction. In ❶, the physical register availability flag is looked up to find a free physical register for \$r0. There can be various architectural to physical register mapping policies depending on the purpose but we assume that the sub-array level power gating is purposed in this case. To power gate as many sub-arrays as possible, the active registers should be consolidated. Therefore, \$P0 that is close to many active registers is selected. The various mapping policies are discussed in Section 7. In ❷, the physical register availability flag is modified so that the bit for \$P0 is set. Then, the renaming table needs to be modified so that the *warp*5 can find \$r0's value from \$P0 in the following instructions. The renaming table entry is found by using combined warp Id and register Id as illustrated in ❷. Finally, in ❸, the \$r0's new value is written to \$P0 at the writeback stage. Note that all the \$P0s in the eight SIMT clusters are written together as the register renaming is done in a warp unit.

The new register allocation occurs only when the architectural register does not have a mapping information. The registers can have no mapping in two cases: 1) at the first access to the architectural register and 2) at the next access after the register release. Therefore, if the illustrated example is done in a diverged flow, the following control flow does not allocate new physical register as the leading control flow already allocated a physical register for \$r0. Therefore, an architectural register for a warp only has one entry in the renaming table.

6.2. Flag Instruction Decoding and Release Flag Cache

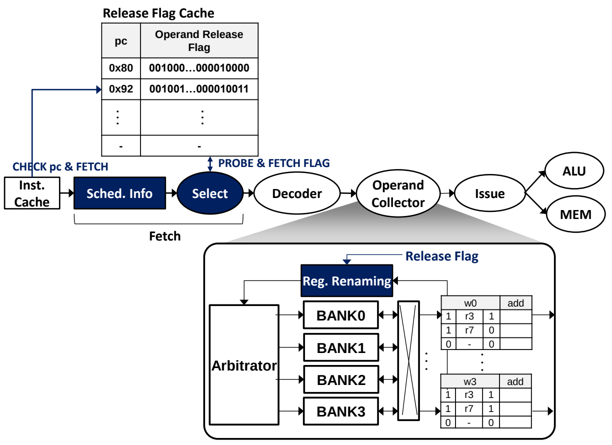


Figure 8: Modified pipeline and release flag cache

To provide the register lifetime information, compiler adds two new release flag instructions. As noted in Section 2.2, a state-of-the-art GPGPU architecture supports similar flag instructions. We assume that there exists a pipeline stage that can recognize the flag instructions before decoding the actual program instructions as described in [17]. The other stages are the same with Fermi. Figure 8 shows the modified pipeine. Fetch stage is partitioned into two: Sched. Info and Select. Sched. Info checks the flag instructions and Select stage chooses one of the program instructions and pushes it to Decoder stage. In the state-of-the-art GPGPU, a metadata instruction is added at every seven instructions. To use the information for the following seven instructions, we believe there should be some sort of buffer to keep the information while fetching the seven instructions. We also use a *Release Flag Cache* that contains the release flags of *pir*. Note that a *pir* keeps the release information for the following 18 instructions. The release flag cache is a shared logic across the warps. As the warps that are scheduled back-to-back are likely to execute the same instruction, they do not need to maintain an exclusive copy of the same *pir*. Instead of fetching a *pir* whenever the program counter reaches to *pir* position, we add a checker logic that selectively fetches the instructions from the instruction cache only when the program counter of the new instruction is not found from the release flag cache. If the *pir* is hit in release flag cache, the instruction is not fetched from the instruction cache and the program counter is incremented to fetch the next instruction. When an instruction is selected in Select stage, the corresponding three bit flag is fetched from the release flag cache by using the instruction's program counter. When the release flag is full, the flag that has greater than 18 instructions worth backward distance from any of the instructions in the instruction buffer is replaced with new flag.

The number of entries in a release flag cache can be less than the maximum number of warps allowed in a SM as a flag is shared by multiple warps. However, to support the case when all the warps execute different program flows, we add as many entries as the maximum number of warps per SM. Each release flag is 54 bits long. As Fermi can have maximum of 48 warps per SM, the total cache size is 324B.

The *pbr* does not need to be stored in the cache as it can simply release the specified registers. When *pbr* is fetched, the register mapping table is looked up and the mapping information is removed and the corresponding bit of the physical register availability flag is cleared.

7. OPTIMIZATIONS

7.1. Renaming Constraints

In our baseline architecture, the renaming table should be 3.69KB to afford all the registers. However, we found from the evaluation that many applications can be serviced by more or less 1KB renaming table. To reduce the renaming table size to 1KB, we set a constraint for register renaming.

The renaming table size can be easily estimated by using the total number of warps and the registers used for the kernel as explained in Equation 1. If the estimated renaming table size is bigger than 1KB, we set a constraint for that workload based on our two observations. First of all, we found that renaming a long lived register is not beneficial since that register cannot be released and reused anyway. Second, if any two registers have the similar lifetime length, the register that has higher write frequency tends to produce less register reuse opportunity since it consumes a physical register more frequently. Therefore, we only allow the registers that are not classified in these two cases.

To do that, we firstly calculate the estimated register value lifetime. The value lifetime can be calculated by the number of instructions in between the value write point and the next release point in the code. Then, the registers are sorted by the lifetime length order. Then, we remove a register from the top in the sorted list and recalculate the renaming table size by assuming that we can rename only the registers remained in the sorted list. This process is repeated until the estimated renaming table size becomes less than 1KB. If there are registers that tie in the lifetime length, we sorted those registers by using their write frequency in the code. According to our second observation, the register that has higher write frequency is removed from the sorted list first. If the renaming table size becomes less than 1KB, the registers remained in the sorted list are considered to be renamed. For the registers removed from the sorted list are assigned a reserved space in the register file and never renamed.

The registers that are exempted from renaming are given different architectural register name in compile time to be distinguished from the renamable registers. Such registers information is encoded to the kernel binary. When the kernel is launched to a SM, the number of renaming exempted registers is decoded from the kernel code binary and a physical register per a renaming exempted register is reserved before starting the program execution. Then, the physical register id for such registers is calculated by using the warp id and the architectural register id at the pipeline decode stage. The bits in the physical register availability flag for the reserved physical registers are set to '1' so that the regular register renaming only happens on the other physical registers.

7.2. Mapping Policy

In the example of Figure 7, $P0 is selected among the available registers because it is in the same sub-array with the other active registers such as $P2, $P3, $P4, and $P5. That way, the number of sub-arrays that can be turned off is maximized. Therefore, for better power efficiency, $P0 is selected. However, the mapping policy can also be tuned for different purposes. For example, the register consolidation policy can cause a wear-out issue as the sub-arrays in which the registers are consolidated tend to be worn out faster. If the reliability is a critical issue, we can make the registers worn out evenly by selecting physical registers from different sub-arrays everytime. The mapping can be easily tuned by remembering the lastly allocated physical register id and then looking up the bits that have a sub-array worth distance with the lastly allocated register id from the physical register availability flag.

Parameter	Value
Simulator	GPGPU-Sim v3.2.1
Execution Width	32-wide SIMT
# Threads/Core	1024
Register Size	128 KB/SM
# Register Banks	32
# Scheduler/SM	2
Scheduler	two_level_active:6:0:1

Table 1: Simulation Parameters

Parameter	Renaming table	Register bank
Size	1KB	4KB
# Banks	4	1
Vdd	0.96V	0.96V
Per-access energy	1.14 pJ	4.68 pJ
Per-bank leakage power	0.27 mW	2.8 mW

Table 2: Register renaming table and register bank energy in 40nm technology

8. EVALUATION

8.1. Settings and Workloads

We used *GPGPU-Sim v3.2.1* [3] to evaluate the proposed register renaming. We assumed a SM has 128KB register file which is partitioned to 32 banks as in Fermi. The two-level-active scheduler is used and the ready queue size is set to six warps. For the compiler, nvcc v4.0 and gcc v4.4.5 is used. Details of the simulation parameters are listed in Table 1. The renaming table and the register bank power parameters are calculated by using CACTI v5.3 by assuming 40nm technology. As described, the renaming table has four banks.

For the workloads, we used several applications from NVIDIA CUDA SDK [4], Parboil Benchmark Suite [5], and rodinia [8]. The number of CTAs, threads per CTA, registers used per kernel, and the number of warps used for the workload are listed in Table 3. The values in the parenthesis of # Regs/Kernel field is the minimum general purpose register count that can avoid register spill. These values are collected by using $-maxregisters$ compile option. The values that are outside of the parenthesis in the same field are the register counts that include the address register and condition register. We used PTXPlus for more realistic register analysis.

We modified the ptx parser code in GPGPU-Sim for analyzing the register lifetime and inserting the two new flag instructions. GPGPU-sim provides a very detailed ptx parsing code that includes basic block recognition and control flow analysis. We traced the source and destination operands of each instruction to figure out the release point. While parsing each instruction of the code, we collected the source operands. Whenever encountering an instruction that uses one of the collected source operands as a destination register, we set the flag bit in the last read instruction. All the bit flags are again collected in 18 instructions or basic block unit to make a *pir* instruction. When encountering a control branch or loop, we analyzed the register access patterns in each diverged flow to

Name	# CTAs	# Thrds/CTA	# Regs/Kernel	# Warps	Name	# CTAs	# Thrds/CTA	# Regs/Kernel	#Warps
matrixMul	50	256	14(7)	32	hotspot	1849	256	28(20)	24
Blackscholes	480	128	18(16)	32	kmeans	1936	256	13(9)	48
dct8x8	4096	64	22(20)	16	NN	168	169	14(8)	42
nbody	60	256	41(40)	32	pathfinder	463	256	16(8)	48
reduction	64	256	14(8)	40	lud	15	32	19(12)	48
vectorAdd	196	256	4(3)	48	gaussian	2	512	8(6)	16
backprop	4096	256	17(12)	48	LIB	64	64	24(17)	8
bfs	1954	512	9(6)	48	LPS	100	128	17(10)	28
heartwall	51	512	29(23)	32	CP	256	128	17(17)	32

Table 3: Workloads

find the proper release point. If any registers are categorized in any of the cases listed in Section 4.3, the release is delayed to the safest merge point.

8.2. Register Savings

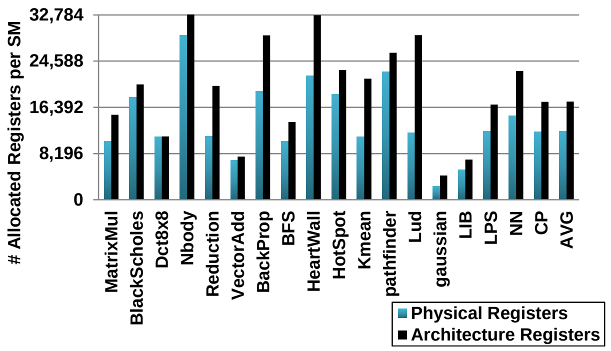


Figure 9: Total number of registers allocated per SM with and without register renaming

Figure 9 shows the total number of registers allocated per SM while executing the workloads with and without register renaming. The bar charts named *Physical Registers* denote the number of registers that are ever allocated by the workloads when register renaming is used. Those named *Architecture Registers* show the number of registers allocated when the normal register allocation is used. In most workloads, smaller number of physical registers are used than the compiler reserved architecture registers when register renaming is used, which leads to average of 30% register savings. The reduced register space leads to a static power efficiency. The power efficiency will be explored in the following section.

Only Dct8x8 shows a very small register savings. This is because, in one of the Dct8x8 kernels, many of the instructions consume a register and produce the register value within an instruction or by the next instruction. In this register access pattern that has frequent but extremely short lifespan, the register renaming is not beneficial because the released register keeps consumed by the warp itself. However, as results plotted in Figure 9 is the total number of registers ever allocated to the register file, much of the allocated spaces are turned off during the execution. Therefore, the workloads like Dct8x8 can still save much power which will be explained in the next section.

8.3. Power Savings over Baseline Power Gating

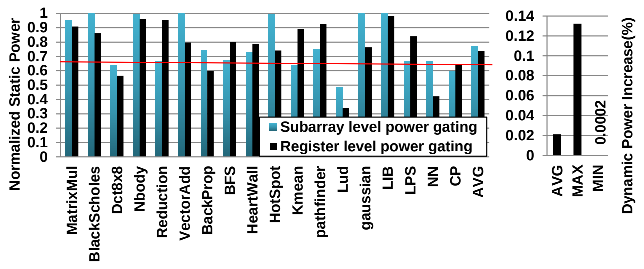


Figure 10: Normalized static power compared to the baseline power gating and dynamic power increase (%) due to renaming table accesses over the register file access power. The red line indicates the expected static power level of the baseline power gating when voltage scaling is used.

We also explored the expected power savings by applying two power gating methods: 1) individual register level power gating and 2) sub-array level power gating. The individual register level power gating shuts down the transistors of a 128-bit register whenever the register is released. To do that, each register should have its own sleep transistor. The sub-array level power gating shuts down whole sub-array when there is no active register in the sub-array. This power gating controls the power in coarser level such that a sleep transistor is used for turning on and off the entire sub-array worth transistors. We follow the power gating models of [6] and [15] for the two power gating methods, respectively. A difference with the models from our model is the number of supply voltage levels. As we know which register is dead every cycle, the registers can be completely shut down. Therefore, we do not assume to use sleep mode that runs by the retention voltage. We do not propose any of the power gating logic itself. Therefore, the sleep mode can be used for more aggressive power efficiency. However, as this paper's purpose is evaluating the power efficiency due to the register renaming, we use simpler assumption that only uses two voltage levels: full Vcc and shut down. The wakeup delay is assumed to be one cycle as in [6, 15]. As the register accesses can be known in the operand collector stage that typically takes two to three cycles to fetch all the registers, the one cycle wakeup delay can be hidden.

Figure 10(a) shows the static power reduction when using register renaming compared to a basic power gating that does

Technology(nm)	$V_{ccmin}(V)$	$V_{cc}(V)$
65	0.7	1.1
45	0.65	1.0
32	0.6	0.9
22	0.55	0.8

Table 4: Retention voltage of SRAM cells from 65nm to 22nm technology nodes [15]

not know the register lifetime apriori. The basic power gating that we compared does not use voltage scaling but it turns on a sub-array or a register only when the sub-array and the register is firstly accessed. As it does not know the register lifetime, once a register is turned on, it cannot be turned off. Note that this is still a valuable comparison because the power savings by using retention voltage level (sleep mode) is getting lower as technology scales as depicted in Table 4. Instead, we put a red line on the plot to show that the expected static power level when falling all the active register cells to retention voltage.

In the applications that use the same number of sub-arrays when with or without register renaming during most of the execution time, such as gaussian, Nbody, and LIB, do not achieve significant power efficiency over the baseline power gating. However, all the other applications effectively reduce the static power by consolidating the active registers into smaller number of sub-arrays. Overall, the average power saving of the register renaming compared to the baseline power gating is 23% in sub-array level power gating and 26% in register level power gating.

We also measured the static and dynamic power overhead of the renaming table as shown in Table 2. A four banked 1KB renaming table consumes 38% of a 4KB register bank static power. Therefore, the total static power overhead due to renaming table is 1.2% as we use one renaming table per SM while there are 32 register banks. The dynamic power overhead of the renaming table is shown in Figure 10(b). The dynamic power of the renaming table and the register file is calculated by accumulating the total accesses to the renaming table and then applying the power parameter. The dynamic power overhead due to the renaming table over the register file is average of 0.02%.

8.4. Renaming Table Size

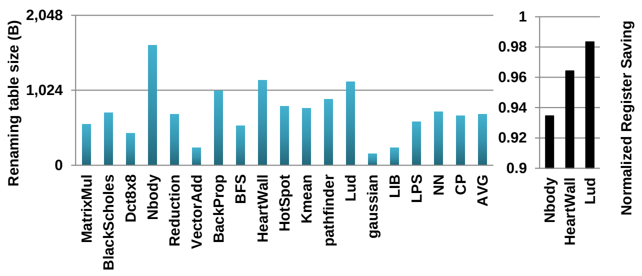


Figure 11: Per SM renaming table size without constraints and normalized register saving when constraint is applied to fit 1KB renaming table

Figure 11(a) shows the renaming table size demanded by the workloads when renaming constraints are not applied. Almost all the workloads used in the evaluation can rename the registers by using 1KB renaming table except the three workloads: Nbody, HeartWall, and Lud. To fit the renaming information into 1KB renaming table, some of the long lived registers of the three workloads are exempted from the renaming by using the value lifetime ranking as described in Section 7. The total number of exempted registers for each workload is 16 among 41 in Nbody, 4 among 29 in HeartWall, and 2 among 19 in Lud. These registers are assigned a reserved physical register and never renamed.

Figure 11(b) shows the register saving reduction over when the renaming constraints are not applied for the three workloads. As expected, Nbody's register saving is reduced the most among them because it can not use renaming for 39% of total registers. However, the reduction scale is less than 1%.

8.5. Static and Dynamic Instruction Increase

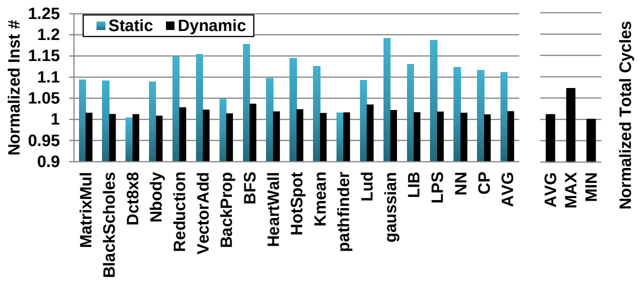


Figure 12: Normalized static and dynamic instruction count and the total execution cycle

The left hand side plot in Figure 12 shows the dynamic and static instruction increase when using register renaming. For the dynamic instruction increase, we compared the total decoded instruction count. As *pbr* and *pir* do not issue any instruction to the execution units, the only overhead that is caused by the two added instructions occurs in decoder logic. However, as *pir* is shared across multiple warps, and *pir* is fetched from instruction cache only when it is not in the release flag cache, the dynamic instruction increase is much less than the static instruction increase. Overall, the dynamic code increase is less than 2% while the code size is increased by average of 11%.

We also compared the execution time with and without register renaming as shown in the right hand side plot in Figure 12. The Y-axis value is the same with the left hand side plot. In the workloads that have relatively short kernel such as BFS, the execution time is increased the most which is 8%. Overall, the timing overhead due to register renaming is average of 1%.

8.6. Scheduler Sensitivity

Figure 13 shows the register saving variance when different warp scheduler is used. Two level scheduler deschedules the warps that have dependency with long latency instructions

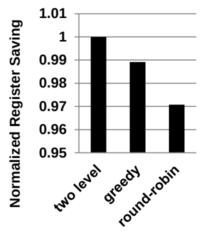


Figure 13: Normalized register savings w.r.t. scheduler

such as *ld* and push the warps to the pending queue. Otherwise, the scheduler takes an instruction from one of the warps in the ready queue. We configured to use six warps per ready queue. Greedy than round-robin scheduler schedules one warp as far as the warp can issue any instruction then move to next warp. The round robin scheduler schedules all the warps interleavingly every cycle.

The register saving is different within 3% range across the schedulers. However, among them, the round robin scheduler derived the worst performance. This is because registers are not likely dead every cycle but as the scheduler moves the warp every cycle, there is less opportunity to reuse any register space released by another warp. For example, there is not an opportunity like Figure 3(b) in the round-robin scheduling because the 'last write' instruction on the \$r12 of all the warps is executed prior to the 'last read' instruction of any warp.

On the other hand, two level and greedy scheduler allocates the registers for a small set of warps then moves to the other warps when the warp cannot proceed any more. According to our experiment, the scheduling period of each warp is from hundreds cycles to thousands cycles depending on the applications. Within the long scheduling period, the register file can maintain only the registers for the small set of warps. And then, another warp can start running after much of the short lived register's space of the descheduled warp is released.

9. RELATED WORK

There have been several studies that address the power consumption concern of GPGPU register file. To reduce the dynamic power, [9] proposed to use small register file cache. They store any newly written register values to a small register cache so that the registers can be read from the cache rather than the huge register file.

In [10], adding to the register cache, two small register files are added. By determining the register location among the two small and one large register files according to the register lifetime, the dynamic power for accessing short lived registers is reduced.

[6] proposed a tri-modal register file structure. By exploiting the fact that register access interval is normally hundreds of cycles, they push the registers to the drowsy mode right after being accessed so that the registers do not burn power for hundreds of cycles before actually accessed again. However, as they do not know each register's lifetime apriori, the SRAM cells for the dead registers should burn power even though it is low power mode. Our work can easily solve this problem by turning off the registers that are dead by using the compile time analysis.

10. CONCLUSION

This paper, we propose a register lifetime aware allocation method. So far, as the register file is accessed in warp unit and regarded as each warp's private resource. Because of that, fairly high portion of register file is unused for a significant amount of time mainly due to the registers that have short lifetime. To reduce the overall live registers thereby decreasing the power consumption as well as the demand for the register file size, we propose to share register file across warps. The compile time register lifetime analysis information is used to release registers from the physical register space immediately after their last use. The released register space is then reassigned to another warp which is just about to start a new register usage. To enable this register reassignment we propose a light weight register renaming in hardware. By releasing registers and reusing them across warps we can reduce the number of concurrently live registers on average by 30%. The reduced live register space leads to an average of 23% and 26% power saving over a basic sub-array level and individual register level power gating.

References

[1] "asfermi: An assembler for the NVIDIA Fermi Instruction Set." [Online]. Available: https://code.google.com/p/asfermi/

[2] "Geforce 400 series." [Online]. Available: http://en.wikipedia.org/wiki/GeForce_400_Series

[3] "Gpgpu-sim." [Online]. Available: http://www.ece.ubc.ca/~aamodt/gpgpu-sim/

[4] "Nvidia cuda sdk 2.3." [Online]. Available: http://developer.nvidia.com/cuda-toolkit-23-downloads

[5] "Parboil benchmark suite." [Online]. Available: http://impact.crhc.illinois.edu/parboil.php

[6] M. Abdel-Majeed and M. Annavaram, "Warped register file: A power efficient register file for gpgpus," in *HPCA*, 2013, pp. 412–423.

[7] R. Alverson *et al.*, "The tera computer system," in *ICS*, 1990, pp. 1–6.

[8] S. Che *et al.*, "Rodinia: A benchmark suite for heterogeneous computing," in *IISWC*, 2009, pp. 44–54.

[9] M. Gebhart *et al.*, "Energy-efficient mechanisms for managing thread context in throughput processors," in *ISCA*, Jun 2011, pp. 235–246.

[10] M. Gebhart, S. W. Keckler, and W. J. Dally, "A compile-time managed multi-level register file hierarchy," in *MICRO*, 2011, pp. 465–476.

[11] S. Hong and H. Kim, "An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness," in *ISCA*, 2009, pp. 152–163.

[12] N. Jayasena *et al.*, "Stream register files with indexed access," in *HPCA*, 2004, pp. 60 – 72.

[13] J. Lai and A. Seznec, "Performance upper bound analysis and optimization of sgemm on fermi and kepler gpus," in *CGO*, 2013, pp. 1–10.

[14] J. Leng *et al.*, "Gpuwattch: Enabling energy optimizations in gpgpus," in *ISCA*, 2013, pp. 487–498.

[15] S. Li *et al.*, "Cacti-p: Architecture-level modeling for sram-based structures with advanced leakage reduction techniques," in *ICCAD*, 2011, pp. 694–701.

[16] NVIDIA, "CUDA Binary Utilities." Available: http://docs.nvidia.com/cuda/cuda-binary-utilities/index.html

[17] NVIDIA, "Nvidia geforce gtx 680 white paper v1.0." Available: http://www.geforce.com/Active/en_US/en_US/pdf/GeForce-GTX-680-Whitepaper-FINAL.pdf