GPU Register File Virtualization

Hyeran Jeon
San Jose State University
hyeran.jeon@sjsu.edu

Nam Sung Kim
University of Illinois, Urbana-Champaign
nskim@illinois.edu

Gokul Subramanian Ravi
University of Wisconsin-Madison
gravi@wisc.edu

Murali Annavaram
University of Southern California
annavara@usc.edu

ABSTRACT

To support massive number of parallel thread contexts, Graphics Processing Units (GPUs) use a huge register file, which is responsible for a large fraction of GPU's total power and area. The conventional belief is that a large register file is inevitable for accommodating more parallel thread contexts, and technology scaling makes it feasible to incorporate ever increasing size of register file. In this paper, we demonstrate that the register file size need not be large to accommodate more threads context. We first characterize the useful lifetime of a register and show that register lifetimes vary drastically across various registers that are allocated to a kernel. While some registers are alive for the entire duration of the kernel execution, some registers have a short lifespan. We propose GPU register file virtualization that allows multiple warps to share physical registers. Since warps may be scheduled for execution at different points in time, we propose to proactively release dead registers from one warp and re-allocate them to a different warp that may occur later in time, thereby reducing the needless demand for physical registers. By using register virtualization, we shrink the architected register space to a smaller physical register space. By under-provisioning the physical register file to be smaller than the architected register file we reduce dynamic and static power consumption. We then develop a new register throttling mechanism to run applications that exceed the size of the under-provisioned register file without any deadlock. Our evaluation shows that even after halving the architected register file size using our proposed GPU register file virtualization applications run successfully with negligible performance overhead.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

MICRO-48 December 05-09, 2015, Waikiki, HI USA

Copyright 2015 ACM ISBN 978-1-4503-4034-2/15/12\$15.00

DOI: <http://dx.doi.org/10.1145/2830772.2830784>.

Categories and Subject Descriptors

C.5.3 Computer Systems Organization [COMPUTER SYSTEM IMPLEMENTATION]: Microcomputers

Keywords

GPGPU, Microarchitecture, Register file, Energy-efficient computing

1. INTRODUCTION

To enable massive thread level parallelism graphics processing units (GPUs) rely on large register files. Register file is the largest SRAM structure on die and the third most power hungry structure [31]. In a GPU, each warp has its own set of dedicated architected registers, indexed by the warp id, and each architected register has a corresponding physical register allocated in the register file [34]. Once a register is allocated it is not released until the cooperative thread array (CTA), which the warp is part of, completes its execution [27]. This policy simplifies register management hardware but at the cost of significant waste of power and underutilization of registers. As power continues to become an impediment to all computing system designs, it is only imperative to look at solutions to reduce GPU register file power.

The goal of this paper is to propose a GPU register file management approach that roots out the inefficiencies in its usage. This work is motivated by our observation that register lifetime, defined as the time from when a new value is written into the register until the last use of that register value, varies by orders of magnitude across various registers in a typical GPU application. The number of live registers at any point in time is much fewer than the total number of registers used in that kernel. Since warps may be scheduled for execution at different points in time, we propose to proactively release dead registers from one warp and re-allocate them to a different warp that may occur later in time, thereby reducing the needless demand for physical registers. CPU designers grappled with the problem of having too few architected registers and how to effectively rename the false dependencies to improve instruction level parallelism. Register renaming was invented

to address this concern and a plethora of enhancements were proposed to improve renaming efficiency [36, 38, 39, 11, 18, 35, 25, 26, 37, 40, 19]. We adapt register renaming for achieving the opposite effect of CPU renaming, namely reducing the number of physical registers without reducing the architected register space. We call this mechanism as GPU register file virtualization.

The main contributions of this paper are:

- We propose a new register management mechanism that uses compiler-generated register lifetime information to proactively release registers from one warp and opportunistically re-allocate that register to other warps. We show that through inter-warp register sharing we reduce dynamic and static power consumption.
- Since inter-warp register sharing reduces physical register pressure we exploit it to design a GPU with half the number of the physical registers, while transparently allowing the applications to use the full architected register space. As the total register file size in a GPU chip is comparable to that of a shared last level cache in a multi-core CPU [3], halving the register file has significant economic and yield impact [45].

2. RELATED WORK AND CHALLENGES OF REGISTER RENAMING IN GPU EXECUTION MODEL

Hardware-assisted eager register release in CPUs: Moudgill et al. [40] proposed a hardware-only method to release dead registers early. The scheme detects how many instructions are going to read a particular architected register before that register is redefined. This value is dynamically computed and stored as a counter associated with each physical register. After each use the counter is decremented and when it reaches zero, the physical register is released. Several approaches used this scheme to reduce power [39], improve reliability [18], and to implement fast checkpointing [38, 11].

Associating a counter for each physical register has very high overhead since GPU physical register file size is much larger; 64K registers in Maxwell GPU [6], versus about 400 registers in Intel Haswell [3]. Furthermore, counter-based reclaiming is ineffective in a GPU due to smaller instruction window size per warp.

Hardware-software cooperative register release in CPUs: Martin et al. [37], Jones et al. [25] and Lo et al. [35] proposed software-hardware cooperative register renaming. Martin et al. [37] uses dead value information (DVI) instructions to release the registers that are dead at the boundary of procedure calls. In Jones' approach [25], compiler identifies single use registers, that are read only once during the execution, and then marks the last use instructions of the single use registers. In Lo's method [35], compiler marks the instruction that lastly uses a register value in CPU to release the register as soon as that instruction is executed. Given that

our work focuses on GPUs there are new challenges and opportunities when renaming is used in the context of GPUs. In a CPU, only one path of a diverged flow is executed, while in the context of GPUs warps traverse all the possible flows sequentially. Hence, GPU register release points differ from CPUs, which is properly dealt with in our work. Furthermore, in [35] they do not consider how to reduce the overhead of release instruction. In our work, we exploit the fact that warps in a GPU execute the same code segment. Hence, the register release metadata instructions that are shared across the warps can be cached to effectively reduce the fetch and decode overhead of these instructions. We also show how early release of physical registers can be used to design a GPU with fewer physical registers without curtailing thread level parallelism. We further curtail the static power consumption by proposing to gate unused register subarrays.

Other orthogonal approaches to improve register efficiency in CPUs: Previous studies leveraged the properties of data stored in registers to improve register file efficiency. Jourdan et al. [26] exploited the value redundancy in the register file to map several logical registers to the same physical register thereby saving physical register space. Ergin et al. [19] proposed register packing inspired by an observation that a large percentage of instructions produce narrow-width results. Lozano and Gao [36] used register renaming to avoid unnecessary commit in an out-of-order CPU by checking if the last use instruction already consumed the data.

GPU register file renaming: To the best of our knowledge, there has not been any study that explored the benefits of GPU register file virtualization. An NVIDIA patent [46] proposed a hardware-only dynamic register allocation and deallocation approach. Once a register space is allocated, the space is deallocated when a new value is written to the architected register. Their approach does not use any compiler knowledge or lifetime analysis. By using register lifetime, we can provide more aggressive register release that leads to less register demand. Moreover, they did not provide any detailed evaluation.

Power efficiency in GPU: To reduce the dynamic power of a GPU register file, Gebhart et al. [20] proposed to use small register file cache. In a following study [21], the authors enhanced the register file cache by adding two small register files. They store short-lived registers to one of two small register files and long-lived registers to a large main register file (MRF). They used the notion of *strand* as a code segment in which all dependencies on long latency instructions are from operations issued in a previous strand. They define a register as long-lived if its lifetime spans across strands. In our work, the compiler-generated information indicates *when a register is safe to be released*. The information is used by the hardware to proactively release registers from one warp and reassign them to a different warp. Rather than relying on a multi-level register file design, we use a traditional one-level register file design. By exploiting warp scheduling differences, we allow regis-

Figure 1: Fraction of live registers among compiler reserved registers captured during the execution (X-axis: cycle, Y-axis: utilization(%))

ter file sharing across warps.

3. BACKGROUND

Metadata instruction: As power efficiency becomes a top design priority, vendors have developed novel approaches to convey compile time information to hardware to improve power efficiency. NVIDIA’s Kepler architecture conveys compile time information to scoreboard logic to track data dependencies. Instead of tracking dependencies at runtime, Kepler relies on compiler to generate the dependency information and this information is conveyed to the hardware using metadata instructions that are embedded in the code [43]. A recent study found that one metadata instruction is added per seven instructions [30], and the format and the operation of the metadata instruction is similar to explicit-dependence lookahead instruction used in Tera computer system [12]. 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. These metadata instructions are fetched and decoded to generate control information for upcoming instructions. To pre-process the metadata instruction, the fetch stage in Kepler 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. *In this paper, we leverage metadata instructions to interface compiler generated information with the hardware.*

GPU register file underutilization: GPU register file is not fully utilized in some benchmarks due to limited parallelism or small code footprint. However, what we observed in this study is that even among the allocated registers only a fraction of those allocated registers are live registers. We define a live register as a register that stores a value that may be consumed by any of the future instructions until the end of program execution. Figure 1 shows the fraction of live registers over all the compiler allocated registers for a sample 10K cycle execution window for a representative set of six applications (more experimental details are presented in Section 9). X-axis denotes time in cycles and Y-axis is

the fraction of all the allocated registers that are alive at a given time. Except VectorAdd, the remaining five applications barely use half of the allocated registers to carry live data. In case of VectorAdd 100% of the allocated registers are live around 2K cycle mark due to the short program code and relatively small number of registers used. Note that while this data is shown for 10K cycles execution window for clarity, there is no significant change in the fraction of live registers used over the entire application execution window. *If one can release the dead registers from one warp and reuse them in another warp then it is possible to build a GPU with a smaller physical register file while transparently allowing the applications to utilize the entire architected register file space.*

It is worth noting that when an application is optimally compiled, all of the allocated registers will be alive at some point during the execution. However, because not all registers are alive during the entire execution time, each warp’s register demand tend to be lower than the peak demand during a large fraction of execution time. Since a GPU executes multiple warps concurrently, the gap between the accumulated architected register file space demand and the actual live register file size necessary to execute these warps grows significantly. In the next section, we propose a register management scheme that exploits register lifetime information and warp scheduling time differences to reduce peak demand and to improve utilization.

4. GPU REGISTER LIFETIME

In this section, we analyze GPU register lifetime characteristics. Figure 2(a) shows three representative register usage patterns seen in GPU applications. The pattern is taken from the benchmark *matrixMul* of CUDA SDK used in our experimental evaluation. The corresponding assembly code is shown in Figure 3. We captured the lifetime of three registers, *r0*, *r1* and *r3*. The X-axis represents time, and the Y-axis represents the register liveness. A dead value is represented with a Y value of zero and live register is represented as a next step up in the Y-axis.

In this program *r1* is written at the beginning of the program and is read at the end of the program execution. As *r1*’s value is read at the end of the program, the register is alive for the entire program duration. Clearly *r1* is a long lived register. On the other hand, *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. This is an example of a register that has multiple lifetimes but each lifetime is relatively short. In this figure, *r3* has the shortest lifetime. It is only used for a short time window at the beginning of the program and the end of the program. After lastly consumed before the loop, *r3* is never used within the loop and then redefined after the loop.

It is worth noting that the register space waste due

Figure 2: (a) Register accesses during the execution time and (b) Expected power efficiency with register renaming

Figure 3: Register lifetime analysis of CUDA SDK matrixMul (LT in the figure refers to register lifetime)

to short lived registers like *r3* cannot be eliminated by compiler optimization because there is a short time window when all three registers are active concurrently. Figure 3 is the compiler generated kernel code from which the register usage patterns shown in Figure 2 are captured. The bidirectional arrows in the figure indicate the lifetimes of the three registers. In two time windows where the code between the offset at 0x90 and 0x108 and between the offset at 0x308 and 0x378 is executed, the three registers are concurrently alive. Therefore, the compiler cannot simply assign *r0* and *r3* to a single architected register.

As shown in this example it is unavoidable to have short lived registers with overlapped lifetime. Even in a single threaded application, wasting a single register is an inefficient use of space. But in a GPU context, all the threads' *r3* registers have the same lifetime pattern. For instance, *matrixMul* assigns 6 concurrent CTAs per SM and each CTA is executed by 8 warps. Therefore, total of 1280 ($6 \times 8 \times 32$) copies of register *r3* are dead for a significant fraction of the kernel's execution time.

5. GPU REGISTER FILE VIRTUALIZATION

We propose a new register management method that effectively reduces the wasted register space. The key idea is to share the register space across the warps by flexibly mapping architected registers to the physical register space. In GPUs, warps are scheduled to execute the same code at different points in time. For instance, using the two-level scheduler [20] that is used in our baseline architecture, the schedule time difference among the warps reaches several hundred cycles because the warps in the pending queue can be scheduled only when the warps in the ready queue encounter long latency memory operations or pipeline stalls. Therefore, when a register's life time ends in one warp, that register space can be allocated to a different warp which is beginning a new register life cycle.

Figure 2(b) shows an example of register reuse. Warp one and three execute the same code but are scheduled in different points in time. Therefore, the short lived register *r3* is used by warp one and warp three in different time slots. If warp one releases *r3* right after the end of its first lifetime (illustrated as white rectangle), warp three can reuse the space for its own *r3* storage.

To enable register sharing across warps, it is necessary to separate architected registers from the physical register space they occupy. CPUs have used register renaming to avoid false data dependency by mapping an architected register's multiple value instances to distinct physical registers. Thus, renaming in CPUs is typically used to enable a larger physical register file than the architected register file. However, in our approach we use renaming to allow multiple architected registers to share a single physical register. This sharing is essentially what virtualization enables. Hence, we call our approach GPU register file virtualization. We will rely on compile-time lifetime analysis to identify dead registers in the code. In the following section we describe how the register lifetime information is collected by simply extending the existing compiler algorithm and how the information is conveyed to the hardware using the metadata instructions.

6. COMPILER SUPPORT

6.1 Register Lifetime Analysis

Intra-Basic Block: The register management logic has to track register lifetime and only when the register is guaranteed to be dead, it can release that register. We will rely on compiler to statically identify the start and end points of the life cycle of each register. 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 flag bit per each instruction operand to indicate whether that register can be released after that read operation. As CUDA instruc-

Figure 4: Register release time w.r.t. lifetime analysis

tions 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. Figure 4(b) and (c) show two scenarios. In both cases, the register can be safely released at the reconvergence point because of the warps’ lock-step execution. In the example of Figure 4(b), the two branch paths are traversed by a warp sequentially. If the release information is put in each flow of the branch, the flow that is executed first will release the registers and the threads within the same warp that execute the other flow may get incorrect results if they use any of the released registers. 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 architected register IDs that can be released at the start of the reconvergence block. The details and overhead are discussed in Section 6.2.

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 statically determine the last use and hence the compiler can release the register only when all iterations are complete. On the other hand, if 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).

6.2 Release Flag Generation

Per-instruction Release Flag: 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 that can be released. If the bit is 1, the corresponding operand register can be released after it is read by the instruction. But embedding a 3-bit *pir* in

Figure 5: Two release flag instructions

each instruction requires significant modification on the instruction fetch and cache access logic. To avoid this concern, we use a 64-bit flag-set meta instruction that is present at the beginning of each basic block as shown in Figure 5(a). The selection of 64-bit flag is to accommodate the fact that CUDA code is already 64-bit aligned. To keep the metadata instruction comply with existing CUDA instruction set that uses 10-bit opcode we simply reserve a 10-bit value as register release opcode, and then use the remaining bits to store 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 metadata instruction 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 [1, 42].

In the example of Figure 5(a), the *pir*’s first three bits represent the release information for each of the input operands of the first *add* instruction. Let us assume that *r0* is determined to be dead after the execution of *add* instruction according to the register lifetime analysis. Since *r0* is the first input operand, the corresponding *pir* flag bit is set to one. The second register operand *r5* 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.

Per-branch Release Flag: 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 5(b), the format is similar to *pir*. The only difference is that every six bits represent 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 released, more *pbrs* are added. However, according to our evaluation, the average number of registers that are released by *pbr* is just 2.

7. ARCHITECTURE SUPPORT

7.1 Renaming Table

To enable register reallocation, each architected register is mapped to a physical register whenever it is writ-

ten. When the architected 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, it is marked as available that can then be remapped to another architected register in the future. The physical register 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 combining warp id and architected register id and contains the corresponding physical register id. In our baseline design, each SM has 128KB register file, which holds a total of 1024 physical registers (32×4 -bytes each); hence, each entry of the renaming table stores the 10 bit physical register number. The total size of renaming table per SM is $(max_warps_per_SM \times max_regs_per_th \times \log_2(max_physical_register_count))$ bits. In our baseline, each SM can support 48 warps and each warp can have a maximum of 63 registers. Thus the total renaming table size is 3.8KB. We will present a simple optimization to reduce the renaming table size later. In addition, we use a single bit register availability vector per each register (1024 bits in total) to indicate if a given physical register is currently assigned or free.

Preserving compiler provided bank information:

The 128KB register file in each SM is divided into four banks. Each bank is further divided into eight sub-banks. Each sub-bank provides data to a four-lane SIMT cluster. Thus a single warp instruction (with 32 SIMT lanes) may access all the eight sub-banks to read one input operand. Since CUDA ISA uses a maximum of three operands each warp instruction may access up to three main register banks. If any of the operands used by an instruction are in the same register bank, bank conflict occurs. GPU compilers strive to avoid register bank conflicts by distributing the input operands across the four main register banks. Kim et al. [27] articulated that the compiler is responsible for reducing the register bank conflict. Lai and Sez nec [30] proposed compiler-level optimizations that show the throughput improvement by reducing the register bank conflicts. Hence, it is prudent not to ignore the compiler allocated bank information while renaming. To preserve the compiler assigned bank information, we restrict register renaming to find a register within the same bank as the original bank that the compiler intended to assign.

To reflect this restriction, we use four 256-bit physical register availability flag vectors; a 256-bit vector per each register bank. In order to rename an architected register, we first identify the bank that the architected register maps to in the absence of renaming, and then restrict searching for an available register within that bank.

Reducing renaming table size: While 3.8KB of renaming table space can completely rename all the available registers in an SM, it is possible to shrink the size of the renaming table by using two observations. First, renaming a long lived register is not beneficial since

Figure 6: Register renaming table and release flag cache

that register cannot be released and reused most of the time. Second, if any two registers have similar lifetime per value instance, the register that has more value instances during the program execution tends to produce less register reuse opportunity since it lives longer by using multiple value instances. For example, if register *r2* and *r3* have the same lifetime, but if *r2* is defined more often in the program than *r3* then we call *r2* as having more value instances. In this case we select *r3* as a short lived register instead of *r2* when we cannot include both registers in the renaming table due to size limitation. Thus register renaming can avoid renaming long lived registers and those registers that have more value instances. Using this approach we can limit the register renaming table size and then select only those registers that benefit most from renaming to fit within the renaming table size. In our approach we limit the renaming table size to 1KB, which is 25% of the full renaming table.

To determine which registers are candidates for renaming, we calculate the estimated register value lifetime at compile time. The value lifetime can be calculated at compile time by counting the number of instructions between the write point and the next release point in the code. Then, the registers are sorted using their lifetime. Once the registers are sorted based on lifetime, the compiler only picks the top $1024B / \{10bits \times \#warps_per_CTA \times \#max_conc_CTAs \times 4B\}$ registers for renaming. Only for the selected registers the compiler inserts *pir* and *pbr* flags to release these registers while non-renamed architected registers are never released. The renaming-exempted registers are assigned the lowest *N* register ids and then *N* is given to the hardware so that the hardware only stores the mapping information for the registers with id that is higher than *N*. The lowest *N* registers are directly mapped to the lowest *N* physical registers, such that each warp's renaming-exempted registers are mapped to *N* physical registers from the register id $N \times warp_id$.

Renaming table organization: The renaming table consists of four banks so that the operands can lookup the physical register index concurrently. When there is a bank conflict, the name lookup may be serialized. The pipeline modification to access the renaming table is illustrated in Figure 6. According to our simulation of this register renaming table, the access latency of the optimized renaming table of 1KB is 0.22ns. In our eval-

uations we conservatively assume that this renaming operation cannot be absorbed in the existing pipeline delays, and hence one extra cycle pipeline latency may be taken for the renaming process.

7.2 Flag Instruction Decoding

To provide the register lifetime information, compiler adds release flag metadata instructions. As noted in Section 3, modern GPUs support metadata instruction decoding for different power efficiency considerations. We rely on the same mechanisms to decode the release flag information. The 10-bit opcode is used to first determine if the instruction type is *pir* or *pbr*. Once this determination is made the remaining 54 bits are simply used to determine which registers are dead when each of the following 18 instructions complete their execution.

Even though the flag instructions can be simply fetched and decoded as normal metadata instructions, to reduce the potential power overhead due to the repeated flag instruction fetches from the regular instruction cache, we use *Release Flag Cache* to store 54 bit flags of *pir* instructions. Figure 6 shows the interaction between register renaming table and the release flag cache. The release flag cache is a direct mapped cache and is shared across the warps. It is indexed by the PC of the *pir* instruction. Multiple warps that are part of a single CTA all execute the same kernel. Since warps within a CTA are scheduled closely in time by most existing warp schedulers, such as round-robin and two-level schedulers, the sequence of instructions executed by different warps exhibit strong temporal locality. Thus, not every warp needs to maintain an exclusive copy of the same *pir*. We add a release flag cache access logic to the fetch stage that selectively fetches the *pir* instructions from the instruction cache only when there is a miss in the release flag cache. Every cycle the fetch stage probes the release flag cache and if the PC is a hit in release flag cache, the *pir* instruction is not fetched from the regular instruction cache and the program counter is incremented to fetch the next instruction. If the instruction fetch width is bigger than one instruction, *pir* may end up getting fetched anyway alongside the regular instructions. Nevertheless, the redundantly fetched *pir* can be easily detected and it is not fed to the decoder logic. In this scenario, while the fetch bandwidth is not altered, at least the decoding cost can be avoided.

When a regular instruction is fetched in select stage, the corresponding three bit flag is also fetched from the release flag cache to determine if any of the operand registers are dead after the current instruction reads the register. The release flag cache is maintained as a direct mapped cache and whenever a PC misses in the release flag cache then the instruction is fetched from the regular instruction cache and decoded. At the decode stage if the instruction was determined to be *pir* then 54 bit release flags are stored in the cache by replacing the existing entry.

To determine the appropriate number of entries in a release flag cache, we measured the dynamic code increase due to *pir* and *pbr* instruction fetching while

varying the size of the release flag cache. According to our results, a release flag cache of 10 entries is sufficient to capture most of the metadata instruction locality. As each release flag is 54 bits long, the total release flag cache size is 68B.

The benefits of caching *pbr* are not significant as these instructions do not control the release of individual register operands of other instructions. Rather they simply release the specified registers. When *pbr* is fetched, the register mapping table is looked up and the mapping information for each released register specified in the *pbr* is removed and the corresponding bit of the physical register availability flag is cleared.

8. USE-CASES

8.1 Register File Under-provisioning

Using renaming, we explore a GPU design that under-provisions the physical register file to be smaller than the architecturally defined register file size needed to support multiple concurrent thread contexts. In particular, our results show that in many applications nearly half of the registers that are allocated by the compiler are dead at any given time. Hence, we propose to design a GPU, called GPU-shrink, with only half the number of physical registers than the baseline; namely, the GPU-shrink has only 64KB register file per SM, compared to the 128KB register file used in our baseline.

Before evaluating the full benefits of GPU-Shrink, we first evaluated power savings from just shrinking the register file. Figure 7 plots the benefits of shrinking the register file, both in terms of reducing dynamic power and static power. These results were generated by using GPUWattch [31] starting with the 128KB banked register file organization as the baseline. Reducing the register file by half reduces dynamic power consumption by 20% and reduces the overall power (leakage and dynamic) by 30%. These results provide the motivation that shrinking the register file size has significant power benefits. However, naively shrinking the register file can lead to significant performance losses. In particular, if the compiler is forced to spill and fill the registers to/from memory the potential power savings will be overrun by the potential latency penalties of accessing memory. Furthermore, the naive approach requires applications to be recompiled to use fewer registers. In the following sections we describe how these potential disadvantages will be alleviated with GPU-Shrink.

First, the unique aspect of the GPU-Shrink design is that it is transparent to the application/compiler layer. The compiler is free to use all the registers in the baseline without any restrictions. The register management hardware simply renames registers using the reduced physical register size. Hence, the availability vector of 1024 registers is now reduced to 512 registers only. If the cumulative live registers across all the CTAs concurrently running on a SM is less than 512, then there is no application perceived difference between GPU-shrink and regular GPU with renaming.

If the live register demand from all CTAs exceeds

Figure 7: Power versus register file size

512 then there will be no available physical registers once all the physical registers are already assigned. To guarantee forward progress we propose that the register renaming logic must reserve a minimum number of registers to allow at least one CTA to make forward progress. When faced with high register pressure the warp scheduler can throttle the register demand by allowing warps from selected CTAs to finish while holding back warps from other CTAs.

We use the following implementation to guarantee progress. The maximum number of registers required for executing a CTA can be obtained from the GPU compiler. For instance, if N is the number of registers needed for each warp and a CTA has M warps then the maximum number of registers required per CTA is $C = N \times M$. The warp scheduler keeps track of the number of registers already assigned to each CTA using a per-CTA register balance counter; a total of eight counters are needed in our baseline architecture since at most eight CTAs can be concurrently executed in an SM. If k_i is the number of registers assigned to CTA_i at a given time then the *counter_i* will store $C - k_i$ as the remaining registers that may be needed in the worst case for CTA_i . Before the warp scheduler selects a warp it checks the number of available physical registers. If the number of available physical registers is greater than the minimum of all $C - k_i$ counter values then it allows the warp to continue. Otherwise, the scheduler recognizes the problem that the available physical registers may be too few to enable at least one CTA to complete its execution. In this case the scheduler simply picks the CTA with the minimum register balance counter (arbitrarily breaking ties) and allows only warps from that CTA to execute; as registers are released by this CTA then new CTAs can again start issuing. Intuitively, the assumption is that a CTA that has already occupied most registers will finish soon or has an opportunity to release more registers than other CTAs.

The above approach avoids deadlocks except for one extremely rare corner case. The CTA level throttling can work effectively unless a kernel assigns only one CTA that requires more live registers than is available in the GPU-shrink. Even though it is a very seldom occurring case (none of our benchmark applications encountered such situation), it is theoretically possible. In this worst case scenario, we rely on conventional register spilling. We rely on the scheduler to automatically issue special spill instructions to a system-reserved memory location when there are not enough registers while running a large CTA. To spill registers the warp scheduler

Figure 8: Effect of register renaming: (a) without power gating (b) with power gating

selects warps in the pending queue. Note that the registers in a warp can be spilled using coalesced memory accesses where registers associated with all the threads in a warp can be spilled by one memory operation per architected register. While the pending warps' registers are maintained in the memory, the active warps will proceed their execution and release as many register as possible. Eventually when more registers than what is required for a pending warp are released the scheduler loads these registers back from memory to physical registers to re-start these warps.

8.2 Static Power Reduction

The register lifetime analysis allows us to power gate all the dead registers. We explored a conventional subarray level power gating approach [32] in the evaluation section. The subarray level power gating shuts down whole subarray when there is no active register in the subarray. Figure 8 shows an example when subarray level power gating is used. The two large blocks in Figure 8(a) represents the two register files, one without register renaming and the other with register renaming enabled, and we show four columns in each block to denote four register banks. 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 subarrays. The left hand side register file shows the active registers distribution when the default register allocation approach is used and the right hand side register file shows register usage when the proposed register renaming is used. By using the register lifetime information given by the compiler, the number of active registers can be reduced. Then, by using the architected register to physical register mapping, the active registers are consolidated into fewer number of subarrays in each bank. Using a single sleep tran-

Figure 9: Leakage under various technologies (P: Planar, F: FinFET)

sistor for the entire subarray in Figure 8(b), all three unused sub arrays can be power gated. Subarray-level power gating can be enabled by simply changing the register allocation policy. Whenever a new register is allocated, we search the available register pool within each subarray range first so that a new subarray is turned on only when the already active subarrays are filled up.

Register file leakage power with FinFET transition: Leakage power is expected to considerably increase every technology generation [28]. Researchers have investigated various device-, circuit-, and architecture-level techniques such as devices with multiple threshold voltages [13, 48, 47], power-gating [22, 29], and voltage scaling [49, 16, 15] to minimize the leakage power consumption. In particular, almost every technology generation has to introduce an innovative device (e.g., high-K/metal-gate strain-enhanced transistors in 45nm technology) just to maintain a constant $I_{off}/\mu m$ [4]; otherwise, it would have been impossible to improve device speed without substantially increasing leakage power. The same goes with the technology transition from 32nm planar MOSFET [44] to 22nm FinFET devices [14]. 22nm FinFET devices improve I_{off}/I_{on} (but not significantly) compared to 32nm MOSFET devices as seen in [14]. The same result is also confirmed in Figure ?? where we use GPUWattch [31]¹ to plot the fraction of leakage power over total GPU chip power when a GPU is designed with 32nm and 22nm MOSFET, and 22nm/16nm/10nm FinFET technologies, normalized to a GPU with 40nm technology. Without the introduction of FinFET devices, a much larger fraction of power consumed by the 22nm MOSFET GPU would have been leakage power. FinFET brings the leakage power back to the baseline, but the climb continues from the new reset point as seen in Figure 9. In other words, minimizing leakage power through various circuit- and architecture-level techniques will continue to be important even in current and future (FinFET) technology generations. Lastly, the GPU register file has been responsible for a large fraction of total power in GPUs (e.g., 15% from our estimation and as shown in [31, 33]).

¹We take PTM [8] for 32nm and 22nm MOSFET, and 22nm, 16nm and 10nm FinFET to update the technology related parameters in GPUWattch with some assistance from the PTM developers. The estimated power consumption numbers also agree with the numbers reported in another more recent GPU power estimation model [33].

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

9. EVALUATION

We used *GPGPU-Sim v3.2.1* [2] to evaluate the proposed register renaming. We assumed that a GPU has 16 SMs and an SM has 128KB register file which is partitioned into four main banks, and each bank has eight sub-banks as in Fermi. Two-level warp scheduler is used and the ready queue size is set to six warps. For the compiler, nvcc v4.0 and gcc v4.4.5 are used. Two schedulers concurrently issue two instructions at every cycle. The renaming table and the register bank power parameters shown in Table 2 are calculated by using CACTI v5.3 by assuming 40nm technology.

For the workloads, we used 16 applications from NVIDIA CUDA SDK [5], Parboil Benchmark Suite [7], and Rodinia [17]. Note that many of the GPU studies using these benchmark suites [9, 23, 24, 10, 41] also used 10-20 benchmarks similar to what we used in this work. They cover a broad range of usage behaviors and provide good insights into the potential benefits of the proposed work, without requiring significant additional simulation resources. The number of CTAs, threads per CTA, registers used per kernel, and the number of concurrent CTAs per SM are listed in Table 1. The value in the parenthesis of # Regs/Kernel field is the minimum required number of registers that can avoid register spills. The values that are outside of the parenthesis in the same field are the register counts that include the address registers and condition registers. We used PTXPlus for 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 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 points for each register accurately.

9.1 Register Size Savings

Figure 10 shows the percentage of reduced register allocations with our proposed approach. We counted the number of physical registers that were actually used (touched at least once) during the renaming process; this is essentially the maximum number of concurrently live registers during any instance in the program execution. We then subtract this count from the total allocated registers to find the total reduced register counts and plotted that a fraction of the total registers allocated by the compiler. Register allocation is reduced up to 44%, and on average 16% of the register space is eliminated from register allocation. Applications with short kernel size (such as VectorAdd) saw smaller reg-

Name	# CTAs	# Thrds/CtA	# Regs/Kernel	Conc. CTAs/Core	Name	# CTAs	# Thrds/CtA	# Regs/Kernel	Conc. CTAs/Core
MatrixMul	64	256	14(7)	6	HotSpot	1849	256	22(20)	3
Blackscholes	480	128	18(16)	8	ScalarProd	128	256	17(11)	6
DCT8x8	4096	64	22(19)	8	NN	168	169	14(8)	8
Reduction	64	256	14(8)	6	LUD	15	32	19(12)	6
VectorAdd	196	256	4(3)	6	Gaussian	2	512	8(6)	3
BackProp	4096	256	17(12)	6	LIB	64	64	22(17)	8
BFS	1954	512	9(6)	3	LPS	100	128	17(16)	8
Heartwall	51	512	29(23)	2	MUM	196	256	19(17)	6

Table 1: Workloads

Figure 10: Register allocation reduction

Figure 11: (a) Performance degradation when using half-sized (64KB) register file and (b) Sensitivity on subarray wakeup latency

ister savings. There is less chance for the dead registers to be reused for other warps due to short execution time. Applications that have longer execution time derive higher register savings and hence our approach is particularly appealing to large kernels.

9.2 Register File Under-provisioning

We compared the performance impact of GPU-shrink with a traditional GPU design that uses only half the number of registers. In the traditional GPU with half the number of registers we force on the compiler to just use half the number of available registers and whenever the compiler needs more registers it has to rely on spilling some registers to memory and later filling them back. We compared GPU-shrink with a simple compiler-enforced register file size reduction mechanism. In our baseline configuration, the applications are already maximally optimized such that the minimum number of registers are used that do not cause any register spill under 128KB register file. Therefore, for this comparison, some applications are recompiled to use less than 64KB registers. Figure 11(a) shows the total execution cycle increase normalized by 128KB register file configuration when GPU-shrink and the compiler en-

forced register file shrink (denoted as *Compiler spill*) are used. Four among 16 benchmarks do not need any throttling because their register usage does not exceed 64KB. Thus, those applications (VectorAdd, BFS, Gaussian, and LIB) had zero performance overhead. In the other applications, GPU-shrink achieves much better performance than simply relying on the compiler to force spills/fills. By releasing and reusing dead registers the effective register file demand is reduced thereby leading to minimal performance overhead. In some applications, the performance is even enhanced when using GPU-shrink; MUM had significant improvement. Further analysis showed that this unexpected behavior is because GPU-shrink dispersed memory contention by throttling some warps leading to performance improvements in those applications. Overall, GPU-shrink suffered 0.58% performance overhead on average, while compiler spill approach suffered 73% increase in execution time.

We also evaluated GPU-shrink-40% and GPU-shrink-30% that uses 40% and 30% smaller register files, respectively. Since our 50% shrink gets zero performance overhead, the additional registers available with these two configurations did not have any impact on the execution latency.

We also measured the performance degradation due to subarray wakeup delay. Figure 11(b) shows the normalized average simulation cycle over when the power gating is not used. We used CACTI-P [32] to measure the wakeup delay for our register file subarray structure. CACTI-P estimated the wakeup delay to be less than one cycle. Nonetheless, for exploration purpose, we used a wakeup delay of 1, 3 and 10 cycles. The performance overheads are less than 2% even with a wakeup delay of 10 cycles. The reason for this low performance impact is that during program execution the number of subarray wakeup events were negligibly small compared to the total execution cycles.

Figure 12 shows the register file energy breakdown of three different design options, normalized to 128KB register file that does not use any renaming. *Dynamic* and *Static* are the register file’s dynamic and static energy consumption. *Renaming Table* is the additional energy consumed by renaming table. *Flag Instruction* includes the energy that is consumed by fetching and decoding release flag instructions and by release flag instruction cache. The fetch/decode energy is measured by GPUWattch. The first bar labeled *128KB RF w/ PG* shows total register file energy consumption when the register file uses sub-array level power gating af-

Figure 12: Total register file energy breakdown

Figure 13: Static code increase and dynamic code increase w.r.t. # entries in a release flag cache

ter applying renaming. This bar essentially shows the energy reduction when we use register renaming just to reduce the static power but do not alter the physical register file structure. The second bar (*64KB RF*) shows the energy savings by cutting the register file into half while using renaming. By halving the register file size, both dynamic and static power can be reduced even without power gating and hence, the average energy saving is even greater than the full size register file with power gating. However, some applications, such as VectorAdd, LUD, Gaussian, and LIB, spend vast majority of execution time on the code segments that have very few live registers. Thus, static power savings are significant when using power gating (as shown in *64KB RF w/ PG*). Reducing the register file size without any power gating actually leads to a small increase in the energy consumption compared to power gating a 128KB register file. But when sub-array level power gating is applied on top of the GPU-shrink, as plotted in the third bar, the overall energy savings increase across all the benchmarks. On average, the GPU-shrink with register under-provisioning and sub-array power gating saved a total of 42% register file energy.

9.3 Static and Dynamic Code Increase

Figure 13 shows the static and dynamic instruction increase when using register renaming due to the new metadata instructions that were added to the code. The dynamic instruction count was measured as the number of instructions decoded by varying the number of entries in a release flag cache. The integer value next to *Dynamic-* indicates the number of release flag cache entries used for the evaluation (i.e. Dynamic-5 uses a

Figure 14: Per SM renaming table size w/o constraints and normalized register saving w/ 1KB constraint

five-entry release flag cache). 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, a new *pir* is fetched and decoded only when it is not in the release flag cache. Therefore, the dynamic instruction increase is much less than the static instruction growth when more entries are added to a release flag cache. Overall, the increased dynamic code (11% without release flag cache as shown by Dynamic-0) is almost entirely eliminated when using a ten-entry release flag cache (0.2% dynamic code increase as shown by Dynamic-10).

9.4 Renaming Table Size

The left hand side chart of Figure 14 shows the renaming table size without constraining the size of the table. Almost all the workloads used in the evaluation can rename the registers by using 1KB renaming table except the three workloads: MUM, Heartwall, and LUD. Our worst case renaming table size estimation assumes 48 warps, each accessing 63 architected registers, but this is only an upper limit that is not reached. Thus when we constrain the renaming table size to 1KB only, these three benchmarks were forced to eliminate a few long lived registers from the renaming process. The total number of exempted registers is 2 out of 19 in MUM and LUD, and 4 out of 29 in Heartwall. These exempt registers are assigned a physical register and were never renamed. The right hand side graph of Figure 14 shows the impact of constraining the renaming table size. Since some of the registers were exempt from renaming a few opportunities to reuse those long lived

Figure 15: Comparisons with hardware-only register renaming [46].

registers (after dead) were lost. As expected, Heartwall’s register saving is reduced the most among them because it can not rename 13% of total registers.

9.5 Comparison With Hardware-Only Approach

We also compared our results with hardware-only register renaming [46]. Hardware-only register renaming approach maps an architected register to a physical register when the architected register is defined. When the architected register is redefined, the mapping is released and the new defined instance of the architected register is mapped to one of the available physical registers. Though [46] was not aimed at power saving, the approach can also be enhanced to save power by power gating unmapped physical registers.

Figure 15(a) shows the register file size reduction of [46] normalized to our proposed approach. In [46], physical registers are released only when the mapped architected register is re-defined. Therefore, the architected to physical register mapping may last even after the architected registers are dead. Thus, waiting for a register to be re-defined is not as effective as our compiler-supported approach, which releases the registers as soon as the lifetime of a register ends.

Figure 15(b) is the register file static power reduction of [46] compared to our approach. In benchmarks such as Heartwall, HotSpot, and LUD, the number of register allocations did not decrease with [46] but that approach can still save some static power. Many of the allocated registers may be defined for the first time only late during the program execution, which allows [46] to power gate the registers before their first definition. However, our approach proactively releases and reassigns registers thereby reducing the total demand on the register file, which saves $2\times$ more static power than [46] when using a 128KB register file. More critically, as [46] does not fundamentally reduce register allocations, it cannot cut the physical register file size without significant performance penalty due to compiler generated spills. Our approach on the other hand can incorporate GPU-

shrink with compiler support, which enables register file underprovisioning thereby saving both static and dynamic power.

10. CONCLUSION

The default GPU register management method allocates a physical register to every architected register and then reclaims these registers at the end of a CTA execution. We showed that a more efficient register management approach is viable if the compiler can provide lifetime analysis of register usage. The hardware uses the register lifetime information to proactively reclaim registers and then reuse those registers across different warps. With reduced register demand, we explored how to manage a GPU with just half the number of registers, while still allowing the applications to use the entire architected registers during compilation.

11. ACKNOWLEDGEMENTS

This work was done when Hyeran Jeon was a PhD student in the University of Southern California. This work was supported in part by NSF (0954211 and 1217102) and DARPA (HR0011-12-2-0020). Nam Sung Kim has a financial interest in AMD and Samsung Electronics.

12. REFERENCES

- [1] “asfermi: An assembler for the NVIDIA Fermi Instruction Set.” [Online]. Available: <https://code.google.com/p/asfermi/>
- [2] “Gpgpu-sim.” [Online]. Available: <http://www.ece.ubc.ca/~aamodt/gpgpu-sim/>
- [3] “Haswell - The 4th Generation Intel®Core™Processor Platform.” [Online]. Available: <http://www.intel.com/content/www/us/en/intelligent-systems/shark-bay/4th-generation-core-q87-chipset.html>
- [4] “International Technology Roadmap for Semiconductors.” [Online]. Available: <http://www.itrs.net/Links/2013ITRS/2013Chapters/2013ExecutiveSummary.pdf>
- [5] “Nvidia cuda sdk 2.3.” [Online]. Available: <http://developer.nvidia.com/cuda-toolkit-23-downloads>
- [6] “NVIDIA Maxwell Architecture.” [Online]. Available: <https://developer.nvidia.com/maxwell-compute-architecture>
- [7] “Parboil benchmark suite.” [Online]. Available: <http://impact.crhc.illinois.edu/parboil.php>
- [8] “Predictive Technology Model (PTM).” [Online]. Available: <http://ptm.asu.edu/>
- [9] M. Abdel-Majeed and M. Annavaram, “Warped register file: A power efficient register file for gpgpus,” in *HPCA*, 2013, pp. 412–423.
- [10] M. Abdel-Majeed, D. Wong, and M. Annavaram, “Warped gates: Gating aware scheduling and power gating for gpgpus,” in *MICRO*, 2013, pp. 111–122.
- [11] H. Akkary, R. Rajwar, and S. T. Srinivasan, “Checkpoint processing and recovery: Towards scalable large instruction window processors,” in *MICRO*, 2003, pp. 423–.
- [12] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith, “The tera computer system,” in *ICS*, 1990, pp. 1–6.
- [13] M. Anis and M. Elmasry, *Multi-threshold CMOS digital circuits-managing leakage power*, 1st ed. Springer, 2003.
- [14] C. Auth, C. Allen, A. Blattner, D. Bergstrom, M. Brazier, M. Bost, M. Buehler, V. Chikarmane, T. Ghani, T. Glassman, R. Grover, W. Han, D. Hanken,

- M. Hattendorf, P. Hentges, R. Heussner, J. Hicks, D. Ingerly, P. Jain, S. Jaloviar, R. James, D. Jones, J. Jopling, S. Joshi, C. Kenyon, H. Liu, R. McFadden, B. McIntyre, J. Neiryneck, C. Parker, L. Pipes, I. Post, S. Pradhan, M. Prince, S. Ramey, T. Reynolds, J. Roesler, J. Sandford, J. Seiple, P. Smith, C. Thomas, D. Towner, T. Troeger, C. Weber, P. Yashar, K. Zawadzki, and K. Mistry, "A 22nm high performance and low-power cmos technology featuring fully-depleted tri-gate transistors, self-aligned contacts and high density mim capacitors," in *VLSIT*, June 2012, pp. 131–132.
- [15] S. Borkar, "The Exascale Challenge." Available: http://cache-www.intel.com/cd/00/00/46/43/464316_464316.pdf
- [16] T. D. Burd and R. W. Brodersen, "Design issues for dynamic voltage scaling," in *ISLPED*, 2000, pp. 9–14.
- [17] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *IISWC*, 2009, pp. 44–54.
- [18] O. Ergin, D. Balkan, D. Ponomarev, and K. Ghose, "Increasing processor performance through early register release," in *ICCD*, Oct 2004, pp. 480–487.
- [19] O. Ergin, D. Balkan, K. Ghose, and D. Ponomarev, "Register packing: Exploiting narrow-width operands for reducing register file pressure," in *MICRO*, 2004, pp. 304–315.
- [20] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron, "Energy-efficient mechanisms for managing thread context in throughput processors," in *ISCA*, Jun 2011, pp. 235–246.
- [21] M. Gebhart, S. W. Keckler, and W. J. Dally, "A compile-time managed multi-level register file hierarchy," in *MICRO*, 2011, pp. 465–476.
- [22] Z. Hu, A. Buyuktosunoglu, V. Srinivasan, V. Zyuban, H. Jacobson, and P. Bose, "Microarchitectural techniques for power gating of execution units," in *ISLPED*, 2004, pp. 32–37.
- [23] A. Jog, O. Kayiran, N. Chidambaram Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "Owl: Cooperative thread array aware scheduling techniques for improving gpgpu performance," in *ASPLOS*, 2013, pp. 395–406.
- [24] A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "Orchestrated scheduling and prefetching for gpgpus," in *ISCA*, 2013, pp. 332–343.
- [25] T. M. Jones, M. F. P. O'Boyle, J. Abella, A. Gonzalez, and O. Ergin, "Compiler directed early register release," in *PACT*, 2005, pp. 110–122.
- [26] S. Jourdan, R. Ronen, M. Bekerman, B. Shomar, and A. Yoaz, "A novel renaming scheme to exploit value temporal locality through physical register reuse and unification," in *MICRO*, 1998, pp. 216–225.
- [27] H. Kim, R. Vuduc, S. Bagsorkhi, J. Choi, and W.-m. Hwu, *Performance Analysis and Tuning for General Purpose Graphics Processing Units*, 1st ed. Morgan & Claypool Publishers, 2012.
- [28] N. S. Kim, T. Austin, D. Blaauw, T. Mudge, K. Flautner, J. S. Hu, M. J. Irwin, M. Kandemir, and V. Narayanan, "Leakage current: Moore's law meets static power," *Computer*, vol. 36, no. 12, pp. 68–75, Dec. 2003.
- [29] S. Kim, S. V. Kosonocky, and D. R. Knebel, "Understanding and minimizing ground bounce during mode transition of power gating structures," in *ISLPED*, 2003, pp. 22–25.
- [30] J. Lai and A. Sezenc, "Performance upper bound analysis and optimization of sgemm on fermi and kepler gpus," in *CGO*, 2013, pp. 1–10.
- [31] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "Gpuwattch: Enabling energy optimizations in gpgpus," in *ISCA*, 2013, pp. 487–498.
- [32] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, and N. P. Jouppi, "Cacti-p: Architecture-level modeling for sram-based structures with advanced leakage reduction techniques," in *ICCAD*, 2011, pp. 694–701.
- [33] J. Lim, N. B. Lakshminarayana, H. Kim, W. Song, S. Yalamanchili, and W. Sung, "Power modeling for gpu architectures using mcpat," *TODAES*, vol. 19, no. 3, pp. 26:1–26:24, Jun. 2014.
- [34] J. E. Lindholm, M. Y. Siu, S. S. Moy, S. Liu, and J. R. Nickolls, "Simulating multiported memories using lower port count memories," in *US Patent No. 7339592*, 2008.
- [35] J. L. Lo, S. S. Parekh, S. J. Eggers, H. M. Levy, and D. M. Tullisen, "Software-directed register deallocation for simultaneous multithreaded processors," *TPDS*, vol. 10, no. 9, pp. 922–933, Sep. 1999.
- [36] L. A. Lozano and G. R. Gao, "Exploiting short-lived variables in superscalar processors," in *MICRO*, 1995, pp. 292–302.
- [37] M. M. Martin, A. Roth, and C. N. Fischer, "Exploiting dead value information," in *MICRO*, 1997, pp. 125–135.
- [38] J. F. Martínez, J. Renau, M. C. Huang, M. Prvulovic, and J. Torrellas, "Cherry: Checkpointed early resource recycling in out-of-order microprocessors," in *MICRO*, 2002, pp. 3–14.
- [39] T. Monreal, V. Vinals, A. Gonzalez, and M. Valero, "Hardware schemes for early register release," in *ICPP*, 2002, pp. 5–13.
- [40] M. Moudgill, K. Pingali, and S. Vassiliadis, "Register renaming and dynamic speculation: An alternative approach," in *MICRO*, 1993, pp. 202–213.
- [41] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt, "Improving gpu performance via large warps and two-level warp scheduling," in *MICRO*, 2011, pp. 308–317.
- [42] NVIDIA, "CUDA Binary Utilities." Available: <http://docs.nvidia.com/cuda/cuda-binary-utilities/index.html>
- [43] 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>
- [44] P. Packan, S. Akbar, M. Armstrong, D. Bergstrom, M. Brazier, H. Deshpande, K. Dev, G. Ding, T. Ghani, O. Golonzka, W. Han, J. He, R. Heussner, R. James, J. Jopling, C. Kenyon, S.-H. Lee, M. Liu, S. Lodha, B. Mattis, A. Murthy, L. Neiberg, J. Neiryneck, S. Pae, C. Parker, L. Pipes, J. Sebastian, J. Seiple, B. Sell, A. Sharma, S. Sivakumar, B. Song, A. St.Amour, K. Tone, T. Troeger, C. Weber, K. Zhang, Y. Luo, and S. Natarajan, "High performance 32nm logic technology featuring 2nd generation high-k + metal gate transistors," in *IEDM*, Dec 2009, pp. 1–4.
- [45] J. M. Rabaey, A. Chandrakasan, and B. Nikolic, *Digital Integrated Circuits (2nd Edition)*, 2003.
- [46] D. Tarjan and K. Skadron, "On demand register allocation and deallocation for a multithreaded processor," in *US Patent No. 20110161616*, 2011.
- [47] L. Wei, Z. Chen, M. Johnson, K. Roy, and V. De, "Design and optimization of low voltage high performance dual threshold cmos circuits," in *DAC*, 1998, pp. 489–494.
- [48] T. Yamashita, N. Yoshida, M. Sakamoto, T. Matsumoto, M. Kusunoki, H. Takahashi, A. Wakahara, T. Ito, T. Shimizu, K. Kurita, K. Higeta, K. Mori, N. Tamba, N. Kato, K. Miyamoto, R. Yamagata, H. Tanaka, and T. Hiayama, "A 450 mhz 64 b risc processor using multiple threshold voltage cmos," in *ISSCC*, Feb 2000, pp. 414–415.
- [49] B. Zhai, D. Blaauw, D. Sylvester, and K. Flautner, "Theoretical and practical limits of dynamic voltage scaling," in *DAC*, 2004, pp. 868–873.