Warped Register File: A Power Efficient Register File for GPGPUs

Mohammad Abdel-Majeed, Murali Annavaram
Electrical Engineering Department, University of Southern California
Los Angeles, CA 90089
{abdelmaj, annavara}@usc.edu

Abstract

General purpose graphics processing units (GPGPUs) have the ability to execute hundreds of concurrent threads. To support massive parallelism GPGPUs provide a very large register file, even larger than a cache, to hold the state of each thread. As technology scales, the leakage power consumption of the SRAM cells is getting worse making the register file static power consumption a major concern. As the supply voltage scaling slows, dynamic power consumption of a register file is not reducing. These concerns are particularly acute in GPGPUs due to their large register file size. This paper presents two techniques to reduce the GPGPU register file power consumption. By exploiting the unique software execution model of GPGPUs, we propose a tri-modal register access control unit to reduce the leakage power. This unit first turns off any unallocated register, and places all allocated registers into drowsy state immediately after each access. The average inter-access distance to a register is 789 cycles in GPGPUs. Hence, aggressively moving a register into drowsy state immediately after each access results in 90% reduction in leakage power with negligible performance impact. To reduce dynamic power this paper proposes an active mask aware activity gating unit that avoids charging bit lines and wordlines of registers associated with all inactive threads within a warp. Due to insufficient parallelism and branch divergence warps have many inactive threads. Hence, registers associated with inactive threads can be identified precisely using the active mask. By combining the two techniques we show that the power consumption of the register file can be reduced by 69% on average.

1 Introduction

General purpose graphics processing units (GPGPUs) use an execution model called SIMT (Single Instruction Multiple Threads) [4] that allows many of the processing elements to share a single program counter to execute the same instruction but on different data elements concurrently. Concurrent thread execution with fast thread switching is supported by a large register file, even larger than a cache, that holds much of the execution state of each thread. For example, in GTX480 GPU there are a total of 16 streaming multiprocessors (SMs) and each SM has 32 cores. To enable 32 concurrent threads, each SM has a 128KB register file. However, each SM has only 16KB L1 cache and 48KB shared memory. Thus the total size of the register file across all SMs is 2MB, while the shared L1 cache size is only 512 KB. The inversion in sizing between cache and register file, compared to the traditional memory hierarchy in the CPU, is a critical microarchitectural feature that is needed for supporting massively parallel execution.

Operating a large register file consumes significant dynamic and leakage power. This problem will get even worse in the future: the reduction in supply voltage has slowed in recent years thereby limiting dynamic power scaling ability of a transistor. The reduction in threshold voltage is leading to a significant increase in the leakage power. As a result, GPGPU register file power consumption is receiving significant attention from the industry and the academic community [13][29].

There has been prior research in reducing the register file power consumption in traditional CPUs [16] [19] [25]. The SIMT execution model, however, provides GPGPU-specific opportunities to further reduce register file power. For instance, our analysis (details presented later) shows that once a register is accessed (read/write) by a thread that register is not accessed again for several hundreds of cycles. The long inter-access delay can be exploited to save leakage power by placing registers in drowsy state immediately after each access. Second, the utilization of SIMT cores within a GPGPU vary dynamically due to the varying amount of parallelism and branch divergence problems in the applications. While underutilization of cores is a concern even in chip multiprocessors (CMPs) the magnitude of underutilization in SIMT cores is much higher due to the massive amount of available resources on a GPGPU. As GPGPUs are increasingly deployed in wide range of application do-

978-1-4673-5587-2/13/$31.00 ©2013 IEEE

mains, the parallelism variance in application activity will only grow. Hence, dynamically disabling access to inactive registers can save significant amount of dynamic power.

In this paper we take advantage of the GPGPU-specific microarchitectural features and application knowledge to introduce two techniques to reduce the leakage and the dynamic power of the GPGPU register file.

Tri-modal register file: We exploit the property that the inter-access cycle count to registers is in the order of hundreds of cycles across a wide range of GPGPU workloads. We propose and evaluate a tri-modal register file that can switch between ON, OFF and drowsy states to reduce the leakage power consumption.

Active mask aware gating: We exploit the dynamic variance in the available parallelism within a warp to disable bitline and wordline activity of unused registers. We rely on GPGPU's built-in active mask feature to identify inactive threads within a warp well ahead of scheduling an instruction. Thus by using the active mask we disable unnecessary register activity to reduce the dynamic power.

By combining the above two techniques we show that the power consumption of the register file can be reduced by 69% on average.

The paper is organized as follows: Section 2 describes architectural features of GPGPUs that we exploit in this work. Section 3 presents some motivational data regarding GPGPU workload characteristics and how CMOS transistor scaling trend can impact register file power consumption in future. Section 4 discusses the proposed tri-modal register file design. Section 5 discusses the proposed active mask aware register gating technique. Our results are discussed in Section 6. We describe related work in Section 7 and we conclude in Section 8

2 GPGPU Background

GPGPU Microarchitecture: The GPGPU microarchitecture varies per vendor; in our paper we used the Fermi[4] microarchitecture as our baseline GPGPU model. Below is a brief description of the microarchitecture blocks that are relevant to this work.

Each SM in Fermi has 32 processing elements (PEs), 16 load store units(LD/ST), and 4 special function units(SFUs) used for special arithmetic computations like sine, cosine and square root. Each SM has its own thread scheduler and register file. In addition, every SM has its own 64KB shared memory that is accessible by all the threads running on the same SM.

The finer grained scheduled unit of work in GPGPUs is called a warp. A warp is a group of 32 threads executing the same instruction with different input operand values. Multiple warps within the same program are grouped together into one cooperative thread array (CTA). Every CTA is assigned to an SM for execution. In order to improve the efficiency and the utilization of the SMs, more than one CTA can be scheduled per SM. When some warps from one CTA are stalled due to long latency operation, warps from other CTAs can be scheduled to keep the SM fully utilized. The maximum number of concurrent CTAs is limited by the SM resources. For example, the number of required registers by each thread, the number of threads per SM and the available shared memory are some hardware constraints that may limit the maximum possible scheduled CTAs.

Figure 1 shows three critical pipeline stages within an SM that are relevant to this work: namely, a two-level instruction scheduling stage, a register file access stage and the execution stage. The two microarchitectural blocks that are proposed for register file power savings are highlighted in the figure: a tri-modal control unit, coordinated mask aware control unit. These two blocks will be described in detail in section 4 and 5.

Two-Level Thread Scheduler: GPGPUs have hundreds of PEs and thousands of software threads that can be mapped to these PEs. Such concurrency in GPGPUs enables them to hide long latency events, like cache miss events, by simply switching from a ready warp to a waiting warp. While there are different thread schedulers proposed, in this work we assume the presence of a state-of-the-art two-level scheduler [13]. In a two-level scheduler all warps that are waiting on long latency events are placed in a pending warp queue. When all the input operands for a warp are ready (i.e. available in the register file) then that warp is moved to the active warp queue by a level one scheduler. The active warp queue holds all the warps whose input operands are available in register file. Then the level two scheduler only issues an instruction from the active warp queue. This approach simplifies warp scheduling since any warp in the active warp queue will find its inputs in the register file without additional waiting.

A separate scoreboard unit (not shown in the figure) is responsible for solving any RAW and WAW hazards between the threads within the same warp. The scoreboard will reserve the destination registers till the result is ready and written back. Each warp has an entry in the scoreboard. This entry has the list of the reserved registers for each warp which indicate that these registers will be updated by instructions executing from that warp. Before issuing any warp, the level two scheduler will check the scoreboard for the dependencies. If it turns out that the operands are ready and can be read from the register file, then the warp will be assigned a collector unit and the access requests will be sent to register file.

Register File: GPGPUs have a large multi-banked register file that is used to manage the execution context of the warps scheduled on an SM by the two-level scheduler. For instance, in Fermi each SM has a 128KB register file.

An instruction warp is scheduled for execution only when the input registers are ready. Even if the register inputs are ready, accessing a large multi-banked register file can take multiple cycles before the instruction can be executed. In order to reduce the latency penalty associated with register reads, an instruction warp that is ready to be scheduled will be assigned a collector unit. The collector unit stores the warp id, the instruction opcode, the source operand register number, the source operand value and a ready bit for each operand. Note that all input operands are already available in the register file when an instruction is scheduled for execution. Hence the ready bit in a collector unit is simply used to inform the scheduler when the register read operation is complete. Thus the purpose of the collector unit is to spread out accesses to the register file to avoid bank conflicts between different warp requests.

When all the operands values are read from the register file, as indicated by the ready bit, the instruction will be issued to the execution stage. Whenever the instruction is issued, the collector unit will be freed and the scheduler can assign a different instruction to that collector unit. To avoid structural hazards, Fermi has 16 collector units: six units are assigned for the warps that will be scheduled for execution on PEs, eight are assigned to SFU-bound warps, and two are allocated for memory instructions.

Traditionally register files in GPGPUs are very wide [23]; a single entry in a register file is 128 bytes wide and contains 32 32-bit operands. Hence one register entry is able to provide the input operand values for all the 32 threads within the same warp. To reduce the access latency of a large register file, it is divided into multiple single ported banks. While there are multiple possible organizations of a banked register file, the most common approach is to distribute the registers associated with a warp across multiple banks. For instance, two registers R0 and R1 used by the same warp may be placed in different banks. This organization allows multiple registers to be read by each instruction in a warp from across multiple banks. Thus each bank needs only be single ported, which reduces the power and design costs.

Each warp has its own set of registers indexed by the warp id. For example, R1 used by the threads in warp 0 is different from R1 used by the threads executing in other warps. It is likely that registers used by different warps can be assigned to the same bank. Hence, the banked register file organization may lead to some conflicts between requests from different warps when they are mapped to the same bank. The collector unit can handle the bank conflicts by acting as a buffer for register reads. In this paper we assume the base register file is 128KB and it is organized into 16 banks and each register is 128 bytes wide. In Section 5.1 we discuss alternative register file implementations and the applicability of our proposed solutions to those designs.

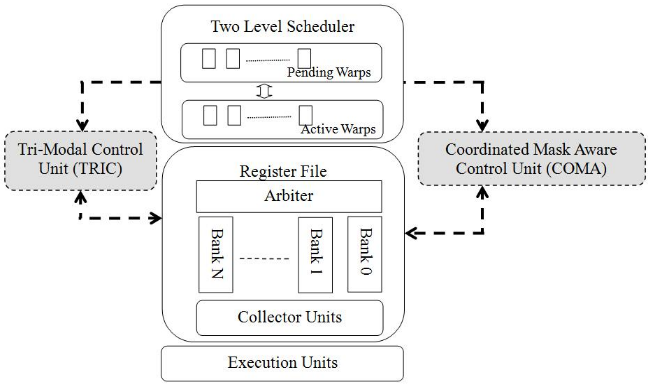

Figure 1. GPGPU Core Pipeline

3 Opportunities for Register File Power Savings

In order to show GPGPU-specific power saving opportunities in register files, in this section we present results from our experiments characterizing several GPGPU workloads. We used benchmarks from NVIDIA Computing SDK[3], Rodinia Benchmark suite [21], and Parboil Benchmark suite[5] . The list of benchmarks used in this study are listed in Column 1 of Table 1. The workload characterization results are obtained by running the benchmark suites using GPGPU-Sim v3.02[8].

Register Allocation at Compile Time: We extracted the number of registers used by the compiler for different benchmark kernels using the *" -ptxas-options=-v"* in the nvcc compilation flags. In Table 1 the last column shows the percentage of the total available registers that are allocated by the compiler for benchmark execution. On average, 46% of the register file is never even allocated for executing a program. Given the vast number of registers available in a GPGPU, compilers simply cannot find enough demand to allocate all available registers for most applications. An unallocated register can be power gated at the beginning of the program execution without worrying about waking up that register. This is just one example of a GPGPU-specific opportunity to reduce register file leakage power.

Register Inter-access Distance: In the next characterization experiment we focus only on the registers allocated by the compiler for a program execution. We measured the number of cycles elapsed between two accesses to the same register. Figure 2 shows the average inter-access cycle count for the allocated registers for several benchmarks. To eliminate the skew generated by a few very low utilized registers, the results shown in Figure 2 exclude the registers with an inter-access cycles count of more than 3000 cycles. Most benchmarks have an inter-access cycles count in the order of hundreds of cycles. On average, once a register is accessed in a cycle its next access will be 789 cycles later.

Benchmarks	concurrent CTAs	Allocated register %
Cutcp	8	62.5%
blackscholes	8	50%
mri-q	6	57%
sgemm	5	53%
Pathfinder	6	43%
streamcluster	2	31.3%
Backprop	5	52%
dct8*8	8	37.5%
nn	8	3.9%
hotspot	3	62.7%
heartwall	2	78.3%
nw	8	9.4%
bfs	3	33.3%
lbm	7	82%
sad	8	43%

Table 1. Workloads' Registers Requirements

When a warp instruction is executed it is unlikely that the same warp is scheduled for execution in the next cycle by the two-level warp scheduler. The only time a warp is scheduled in two consecutive cycles is when no other warp is ready in the active warp queue and the current warp's next instruction is ready for execution. In all other cases, there is a delay between two consecutive scheduling cycles for any warp, which results in large inter-access delay for a given register. This large inter-access delay provides additional leakage power savings opportunities by using drowsy-cache approach to put a register to drowsy state immediately following the current access.

Underutilization of Warps: As mentioned earlier, the minimum unit of work that can be scheduled on a GPGPU is called a warp. Each warp consists of 32 threads executing the same instruction (PC) in a lock-step manner. A fully utilized warp has 32 active threads executing one instruction at a time. Figure 3 shows the breakdown of the number of active threads for different benchmarks. Each bar in the graph is divided into 32 groups. The top most component of a bar *labeled 32* corresponds to the amount of time a warp has 32 active threads compared to the total application execution time. Similarly, the bottom most component *labeled 1* corresponds to the amount of time a warp has only one active thread. We grouped the benchmarks based on active threads count into three categories : Category 1 has the benchmarks where all the warps have 32 active threads throughout the entire benchmark execution. Category 2 has the benchmarks that have utilization levels between 90%-99%. Category 3 has the benchmarks that have utilization levels below 90%.The data shows that only four benchmarks are in Category 1 indicating that many benchmarks rarely utilize all 32 threads. Throughout the rest of this paper, we organized our benchmarks as follows: in all the figures(tables) the left most(upper) group presents Category 1 benchmarks, the group in the middle presents Category 2 benchmarks and the right most(lower) group represents Category 3 benchmarks.

The number of active threads within each warp can be less than 32 for two reasons: First, the benchmark itself may not have enough threads to fill all the warps with 32 threads. The inherent limitation in the amount of available parallelism in a benchmark has not been a significant concern for purely GPU-oriented workloads. But as GPGPUs are used for more general purpose computing the parallelism limitation is becoming a concern. For example, nn benchmark has only 16 active threads in all scheduled warps. Second, if a branch instruction is encountered during the execution, then some of the threads will diverge to the taken path and the rest will diverge to the not-taken path. As a result, the warp will be scheduled in two phases. In the first phase the threads in the taken path will execute and all the threads in the not-taken path will be idle. In the following phase all threads in the not-taken path execute while the threads in the taken path idle. The GPGPU scheduler uses the active mask, a 32-bit vector, that shows the state of the active threads within the scheduled warp in that cycle. If the active mask bit for a thread is zero then that thread will not be active during that cycle.

Even though many warps have fewer than 32 threads, each warp reads all 32 register operands from the register file, which wastes dynamic power. Hence, using active mask to reduce register file activity can reduce dynamic power.

Impact of Register Usage on Power:The data from the characterization experiments showed that in many applications the compiler cannot even use all the available registers and hence many registers are left unallocated. Even when a register is allocated the distance between two consecutive accesses to the same register is around 789 cycles. Thus vast majority of the GPGPU registers are in idle state for long periods. Despite the fact that registers are not accessed, these registers still burn leakage power.

In order to quantify the leakage current effect on the total register file power consumption, we performed a circuit level simulation on a 6T SRAM cells built using 90, 65,and 32 nm technologies. We used the technology files from the predictive models[1]. The leakage current for each SRAM cell is shown in Table 2. The leakage current is measured by measuring the total current drawn from the Vdd when the SRAM cell is in the standby mode(BL,BLB =vdd , WL=0, Data=0, DataB=vdd). The second column shows the supply voltage, the third column quantifies the leakage current of a single SRAM cell and the fourth column quantifies the leakage power of an 8kB register file bank calculated using the technique proposed by [20] . As shown, the Leakage

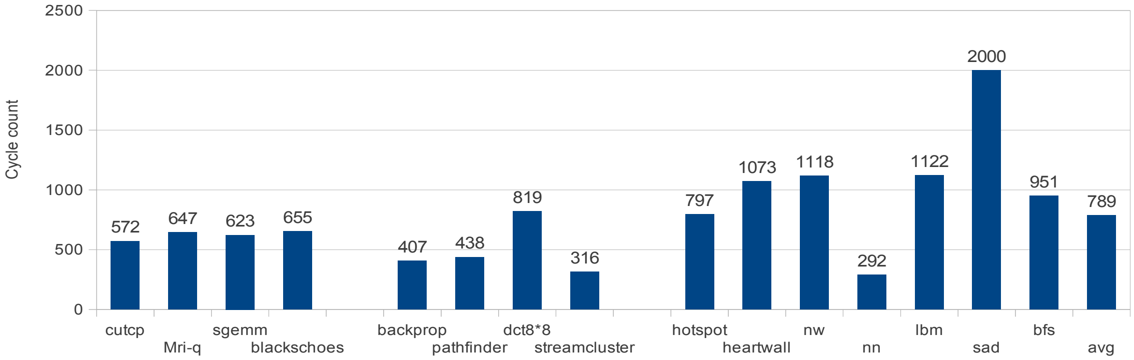

Figure 2. Registers Inter-access Cycle Count

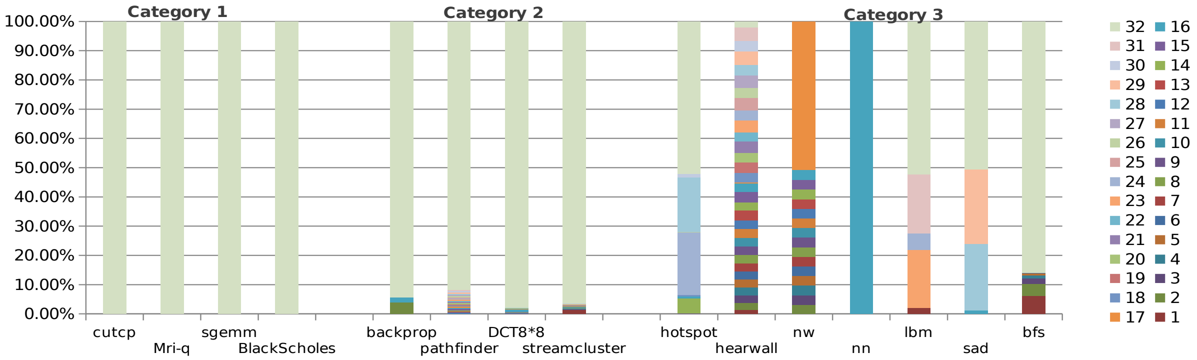

Figure 3. Warp Utilization Breakdown

current nearly doubled as the devices are scaled from 90nm to 32nm.

Technology	VDD (V)	Leakage (nA)	Bank Leakage (mW)	DRV (mV)
90nm	1.2	14.6	2.9	120
65nm	1.1	22.8	3.5	145
32nm	.9	26.08	5.6	220

Table 2. SRAM Leakage Current and DRV Scaling Trend

Static Noise Margin (SNM): As technology scales SNM degrades. As a result, the minimum data retention voltage(DRV) should increase. We measured the DRV for the 6T SRAM cells scaled from 90nm to 32 nm. Table 2 fifth column shows the ideal DRV values for 90nm, 65nm, and 32nm[1]. The retention voltage necessary to keep the data alive in an SRAM cell has increased from 120mV to 220mV.

4 Reducing Register file Leakage Power

Given the opportunities presented in Section 3 we now present our leakage power reduction technique and the required circuit and micro-architecture level support.

Turning OFF Unallocated Registers: Our analysis showed that many registers in GPGPUs are not even allocated for program execution. The first step is to identify all the unallocated registers by analyzing the compiled code. This is a simple static code analysis that can be done on the application binary. We then turn off the unallocated registers completely at the start of the application execution. The microarchitectural support necessary to turn off the GPGPU registers is described later.

Drowsing Allocated Registers: The register inter-access cycle count shown in Figure 2 indicates that there are opportunities to further save on leakage power by exploiting the long idle times between two consecutive accesses to the same register. While turning off the register means zero leakage current, the content of the register is lost. Since threads use registers to maintain the context it will not be possible to just turn off any allocated registers, even if the inter-access delay is long. Hence, we use drowsy mode operation [12] to put the registers into drowsy state. Drowsy state incurs less leakage saving but the register content is saved and will be accessible when needed.

While an ideal drowsy mode operation can operate at DRV, in practice it is necessary to add safety margins to take into account the non-idealities and the mismatch between the transistors. Figure 4 shows the leakage current savings when adding different safety margins. The Y-axis plots the percentage of leakage current consumed by an SRAM cell operating at DRV+margin, compared to an SRAM cell operating at Vdd. The DRV and Vdd values for each technology node are taken from Table 2 . Even if we conservatively add 250mV to DRV an SRAM cell in the drowsy mode consumes less than 10% of the leakage current consumed by the SRAM cell that operates at full Vdd.

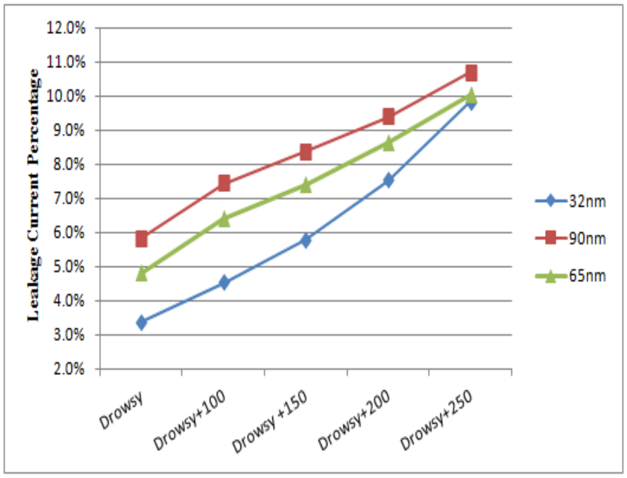

Figure 4. SRAM Cell Leakage Current in Drowsy Mode with Different Safety Margins Normalized to Vdd Leakage Current

Switching Policy: The switching policy decides when to turn ON a register and when to put a register to drowsy state. In this paper, we describe one switching policy called *read-aggressive and write-conservative*. Whenever a specific warp is assigned to a collector unit, the associated input and output registers for that warp are switched from drowsy state to the ON state. The two-level scheduler allocates a collector unit to the scheduled warp and at the same time the switch to ON signal will be sent for the input and output registers. In the most optimistic scenario input registers are read in the next cycle into the collector unit. Due to bank conflicts sometimes the collector unit is unable to read the second input register operand concurrently with the first input operand. Hence, there is a delay between reading the first and second input operands. Also, sometimes even if the warp operands are ready it can not be issued to the execution stage directly because more than one warp(collector unit) can be ready at the same time; we can issue only one warp each cycle. In our proposed policy all input registers of an instruction are woken up as soon as the collector unit is assigned. However, these registers switch back to drowsy state independently as soon as they provide data to the collector unit.

For the output register we use a slightly conservative approach. The output register is kept in the ON state from the time the collector unit is assigned until the time instruction completes execution and writes back the result. The scoreboard will track the output registers for each issued warp already. Once the result is ready and written back to the register file, the scoreboard can instruct the output register to be switched back to the drowsy state.

It is interesting to note that the destination register is likely to be used soon by one of the next few instructions in the warp. Even when the definition to use distance is small in terms of instructions in the warp, the time between executing two instructions in the same warp is quite large since we have a large number of warps. Hence, it is more power efficient to put the register into drowsy state even if the instruction that consumes the register value is immediately after the current instruction in the warp.

4.1 Architectural Support for Tri-mode Operation

Based on the description above, each register entry in the register file needs to be in one of three states: namely ON, OFF and drowsy. In [12], a drowsy control signal is used to switch the cache line between the drowsy state and the ON state. To switch between three states the microarchitectural support suggested in [12] is inadequate for our approach.

In [24] the authors proposed a tri-mode switch that can put the logic in one of three states ON, OFF and drowsy. The proposed switch uses MTCMOS transistors to control the voltage levels and speed. We use the tri-modal switch to place the unallocated registers into OFF state and all the remaining registers are placed either in drowsy or ON state. As described in our switching policy we place the register in drowsy state by default. Whenever the register is accessed we move that register to the ON state temporarily to enable register read/write accesses. The register will switch back to drowsy state after the operation is complete.

Figure 5 shows the block diagram of the register file tri-modal control (called the TRIC) unit. The figure shows two rows of registers and each row has 32 32-bit registers, named R_0T_0 indicating Register R0 for Thread T0 and so on. Each register has its own tri-modal switch that receives its control signals form TRIC. TRIC will first receive the application's register allocation information. This information is first extracted by the compiler analysis and is provided as

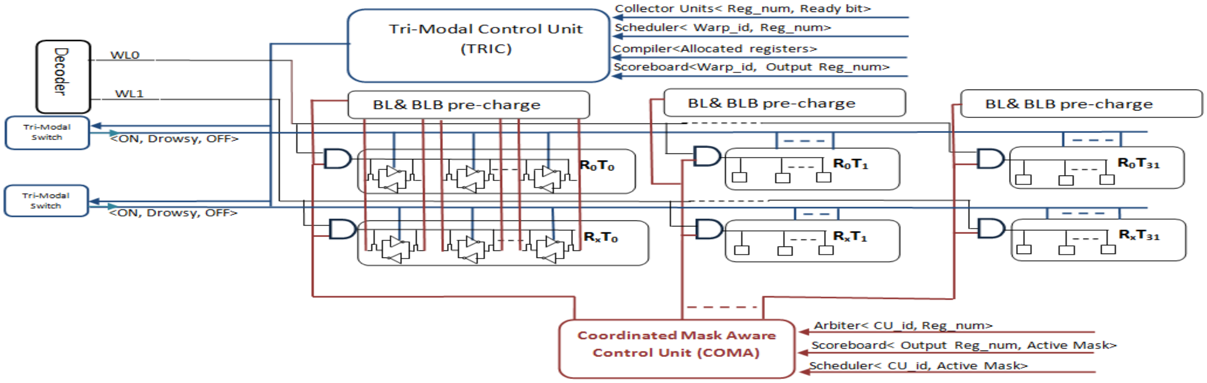
Tri-Modal Control Unit (TRIC)
Collector Units< Reg_num, Ready bit>
Scheduler< Warp_id, Reg_num>
Compiler<Allocated registers>
Scoreboard<Warp_id, Output Reg_num>
Decoder
WL0
WL1
Tri-Modal Switch
<ON, Drowsy, OFF>
Tri-Modal Switch
<ON, Drowsy, OFF>
BL& BLB pre-charge
BL& BLB pre-charge
BL& BLB pre-charge
R0T0
R0T1
R0T31
RxT0
RxT1
RxT31
Coordinated Mask Aware Control Unit (COMA)
Arbiter< CU_id, Reg_num>
Scoreboard< Output Reg_num, Active Mask>
Scheduler< CU_id, Active Mask>

Figure 5. The Proposed Register File with the Tri-Modal Control Unit(TRIC) and the Coordinated Mask Aware Control Unit(COMA) Integrated.

part of the application binary metadata. TRIC will use the allocation information to turn OFF all the unallocated registers at the start of the application execution and then put all the allocated registers into drowsy mode by default.

At run time the two level scheduler sends the input and output registers of the warp that is being assigned a collector unit to TRIC. TRIC will use this information to move input and output registers from drowsy to ON stage. Once a register value is read, the collector unit sets the corresponding input ready bit to ”1” and also informs TRIC that it received the register value. TRIC then switches that input register back into drowsy state. Finally when a warp enters the write back stage the scoreboard sends notification to TRIC that an output register is written. TRIC then switches the output register to drowsy state.

4.2 Architectural Support for Reducing Drowsy Wakeup Latency

One potential drawback of using the drowsy technique is the performance loss due to the wakeup latency. As explained before, the scheduler only looks at the ready warp queue to issue a warp for execution. In our baseline implementation the scheduler sends wakeup notifications to TRIC for registers associated with the warp that is currently being scheduled which only gives one cycle lead time. Thus there is a one cycle delay between when the scheduler sends a wakeup signal to the registers and when the collector unit starts reading the register value. Hence, a one cycle wakeup delay does not cause any performance loss in our base machine. If the wakeup latency is multiple cycles then there may be some performance penalty.

In this section we present one potential approach to entirely hide the multi-cycle wakeup latency of a drowsy register, albeit with a small modification to our baseline. For instance, if drowsy wakeup latency is two cycles, then the scheduler needs to send the register information to TRIC at least one cycle before issuing the associated instruction to the collector unit. To handle this case the scheduler can issue one instruction and concurrently look at the ready warp queue to find the warp that is going to be issued in the next cycle. If the scheduler knows which warp will be issued next cycle it can then pro-actively send the register read information to TRIC one cycle before the warp is scheduled. Thus one can eliminate a two cycle delay associated with wakeup of a drowsy register. In fact the scheduler can look ahead into the ready warp queue to identify the warps that will be issued ”n” cycles ahead and may wakeup the drowsy registers to hide up to ”n” cycles of wakeup delay. In the worst case, the scheduler will only know which warp will be issued only during the start of the cycle. In this case the scheduler cannot hide the latency of register wakeup. In our results section we explored the performance impact of a range of drowsy cache wakeup latencies, assuming there is no way to hide wakeup delays beyond one cycle.

5 Reducing Dynamic Power with Active Mask Aware Gating

In the previous section we described our approach for reducing the leakage power of the register file. But every time a register entry is placed in ON state all 32 register operands associated with a single warp instruction will be woken up. Thus, every access to the register file will read 128 byte register entry to feed the 32 threads in a warp with their source operands. Reading such a large register will incur significant dynamic power because of activating the bitlines, wordline and the sense amplifiers.

As shown in Figure 3, the number of active threads within the scheduled warp are fewer than the warp width of 32 threads. Some benchmarks like nw and nn do not have more than 16 active threads throughout the execution time. During runtime many benchmarks have varying number of active threads within a warp. Scheduling a warp with partial utilization still activates the wide register file entry. For example, scheduling a warp with 31 active threads out of 32 means that we have to charge the wordline segment of 32 cells, pre-charge 32 bit lines (BL) and 32 bit line bars (BLB), and activate 32 sense amplifiers, although only 31 of them are useful for warp execution. According to [17], the active power of read operation from the SRAM cell is proportional to the number of the accessed bits and the access time.

The power optimal solution for such an access behavior is to access only the registers associated with active threads within a warp. GPGPUs already use an active mask to determine which threads are active and which threads are inactive. Hence, we will exploit this information to disable register activity associated with inactive registers. Thus, we will use the active mask of each warp to disable the BL, BLB, sense amplifiers and the output circuitry of the inactive part of the accessed register.

5.1 Architectural Support

Recall that in our baseline design the register file is banked and each register entry is 128 bytes wide. In order to support active mask aware access to the register file, we use the Divided Word Line (DWL) approach [28]. DWL was originally implemented to save dynamic power on large caches where a single word need to be accessed at anytime.

Divided Word-Line(DWL): In the DWL technique the WL is divided or segmented into different wordlines. Figure 6 shows the schematic for the DWL. Each wordline segment has its own local decoder, a simple AND gate, that enables or disables accessing the SRAM cells attached to that WL segment. For our work we modified the original DWL approach so that each access to register entry can provide data to only a subset of threads within a warp. GPGPU designs are particularly suited for easy integration of the DWL approach into a register file. A warp's active mask provides all the necessary decoding information to identify active and inactive threads within a warp. Whenever a register file is accessed, the active mask of the scheduled warp can be used by DWL to activate or deactivate the BL and BLB pre-charge, wordline segments, sense amplifiers and the output circuity.

In order to manage the gating signals we added the coordinated mask aware (COMA) control unit to the register file. The bottom part of Figure 5 shows the block diagram of the register file with COMA integrated. When a read or write operation is issued, COMA loads the warp specific active mask. It then generates the appropriate control signals that can be used to gate the inactive registers. COMA consists of a 16 entry mask (same as the number of the collector units) table indexed by the collector unit entry id. Whenever the two-level scheduler assigns a warp to a collector unit the associated active mask will be written in the designated entry in the mask table. Every register access request is routed to COMA along with the collector unit entry id. The active mask table is then accessed to read the active mask values. The active mask will then be fed to the gating logic to generate the appropriate gating signals. The total size of the active-mask table will be 16*32 = 512 bits. For the output register, the scoreboard will send the active mask of the output register to COMA when the scheduled instruction reaches the writeback stage.

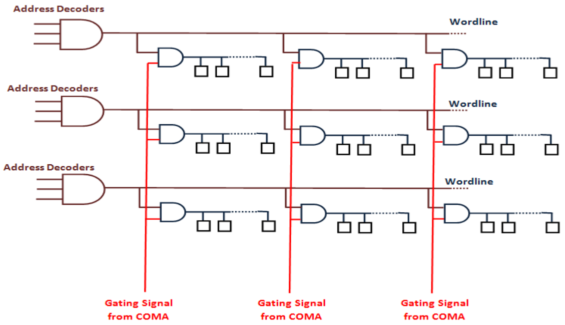

Figure 6. Schematic of the Divided Wordline

Power Efficiency of Modified DWL: In order to quantify the benefits of the DWL technique, we built a 128 byte register entry in Cadence. The register is implemented using the technology files for 90nm, 65nm and 32nm [1]. We extracted the wordline resistance and the capacitance per unit length for different technologies[2][10][15]. Also we estimated the SRAM cell area as $146\lambda^2$[2] where λ is the feature size. The developed RC Model is augmented between every two cells in the 128 byte register. Even after accounting for the additional delay in the AND gate, the reduced RC effects of a long wire of the DWL approach result in a 55%, 31% and 23% reduction in the wordline charging delay for 90nm, 65nm and 32nm technologies, respectively.

Different register file organizations : There are other possible register file organizations than using a single wide register entry. One possible implementation is the one used by [13]. In that organization, the PEs are clustered into groups of four. Each cluster has its own register file which is only 16 bytes wide. The narrower register file provides four 32-bit values to the four associated PEs. Another design option used in [29] splits registers into 32 banks where each bank is just one word wide (4 bytes) and each bank provides data only to one thread within the warp. In fact, every bank is statically assigned to provide data to only one processor element. Irrespective of the register file organization the fundamental problem of activating all 32 registers associated with a single warp still remains. With 32 register banks there is no need for DWL since the wide register is already split into multiple banks. However, even if DWL is not necessary for narrow width registers, the COMA unit can still be repurposed to gate the bank accesses based on active mask.

6 Evaluation

6.1 Simulation Setting and Workloads

We evaluated our proposed techniques for leakage and dynamic power saving techniques using GPGPU-Sim v3.02[8]. We performed our experiments using a Fermi-like GPU configuration. The simulator configuration parameters are shown in Table 3. For the benchmark selection, we covered different programming styles by selecting benchmarks from different benchmark suites. The benchmarks cover different scientific and computation domains that try to benefit from parallel architectures. We used benchmarks from NVIDIA CUDA SDK[3], rodinia Benchmark suite[21] and Parboil benchmark suite[5]. The list of benchmarks used are listed in Table 1.

Hardware Model	Fermi
Execution Model	In-order
no. of SM cores	16
no. of PE per SM core	32
Register file size	128 kB
Register Width	128 Bytes
no. of Banks	16
Warps/SM	48
Warp Scheduler	2-level Scheduler
PTXPLUS	Enabled

Table 3. Simulation Parameters

6.2 Leakage Power Savings with TRIC

In this section we will evaluate the leakage power savings when using TRIC as discussed in Section 4. Using our read-aggressive,write-conservative switching policy, registers spend significant amount of time in drowsy mode. The unallocated registers are of course entirely turned OFF by TRIC. For the results presented in this section we assume a wakeup latency of three cycles, we also assume that during the wakeup duration the register is operating at full Vdd. The first bar in Figure 7 shows the leakage power saving as a percentage of total leakage power consumed without TRIC which is the baseline.

6.3 Dynamic Power Savings with COMA

The second column in Figure 7 shows the power saving as a percentage of total dynamic power consumed without COMA which is the baseline. The dynamic power savings using the COMA unit depends on the activity of the running benchmark. Using the categories presented in the motivation section, it is clear that Category 1 benchmarks will not benefit from this technique since 100% of warps have 32 active threads. On the other hand, the savings obtained from applying the technique on Category 2 and Category 3 benchmarks depends on the number of active threads. Some benchmarks in Category 2 have limited power savings because most of the scheduled warps have only a few (0,1 or 2) inactive threads. As a result, the power saving range from 1% to 6%. Benchmarks such as nw and nn have a large dynamic power savings because they have only 16 active threads out of 32 in all the scheduled warps. Benchmarks such as heartwall have varying amount of thread level parallelism. But COMA can dynamically adjust the number of registers that are turned ON thereby reducing the dynamic power by 46% compared to the baseline. The average dynamic power saving through all Category 2 and Category 3 benchmarks is 19%.

6.4 Combined Savings from TRIC and COMA

In this section we will discuss the results when both TRIC and COMA are applied together on the register file. The combined system first uses TRIC to decide which register entry to bring to active state from drowsy state. Once the register is brought into active state COMA is used to decide which registers to activate for that given warp based on the active mask.

For computing the relative importance of dynamic and static power, we measure the ratio of the leakage and dynamic power of the register file. We used the method proposed by [20] to measure leakage and the dynamic power in the register file organization under study. We extracted the SRAM cell dimensions and the bitline and the wordline capacitances from [2] and [15]. The simulation results show that the dynamic power for reading or writing a 128 byte register is twice the leakage power of the register file bank in 32nm technology.

The third column in Figure 7 shows the total power savings as a percentage of the total power. Total power is computed assuming dynamic power is twice the leakage power. Category 3 benchmarks have the highest power savings because they take advantage of both the leakage and the dynamic power saving techniques. On the other hand, category 1 benchmarks take advantage of only the leakage power saving technique and do not gain from the dynamic power technique. As a result their power saving is less than that of Category 2 and 3 benchmarks.

We also computed total power savings assuming higher dynamic to leakage power ratio. The results show that on

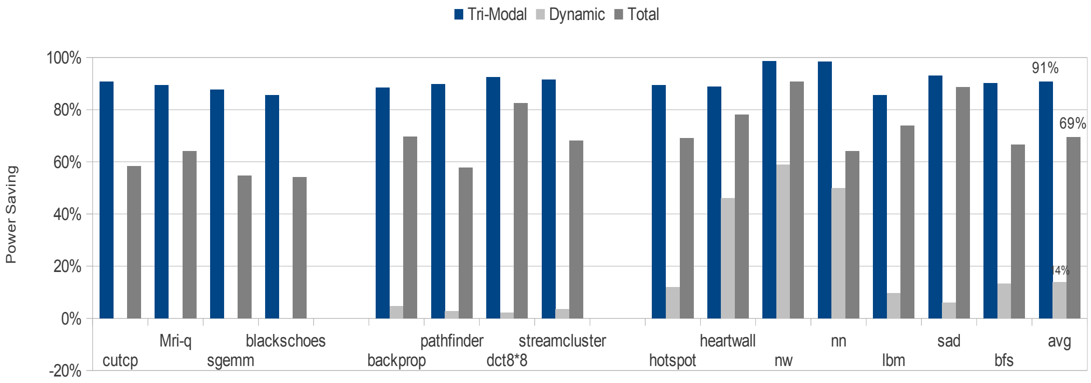

Figure 7. Leakage Power, Dynamic Power and Total Power Savings

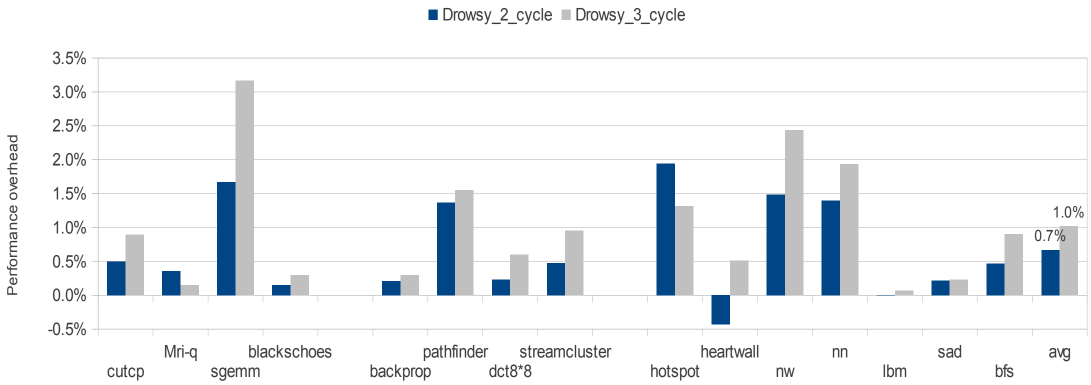

Figure 8. Performance Degradation with Drowsy Wake-up Latency of 2(Drowsy_2_cycle) and 3(Drowsy_3 _cycle) Cycles.

average the total power savings are 69%, 59% and 51.5% when the ratio of dynamic power to leakage power is 2 to 1, 5 to 1 and 8 to 1 respectively.

6.5 Area and Performance overhead

The area overhead in our proposed techniques comes from the AND gates added to the register file to implement DWL, the COMA unit and the associated mask table, and the TRIC unit. The area overhead for all the added components is around 4% the size of the total register file area.

As mentioned earlier, a one cycle wakeup latency for a drowsy register does not impact performance in our baseline since there is a one cycle slack between instruction scheduling and register read operation. In case the register wakeup latency is more than one cycle, we make the worst case assumption that wakeup latency delays a warp's execution. To quantify the effect of the wakeup latency on performance, we ran our benchmarks with two and three cycles wake-up latencies. Figure 8 show the performance degradation with two and three cycles wake-up latency. The results show an average performance degradation of 1.02% in the case of three cycles wake-up latency. Since GPGPUs have large number of ready warps even if a warp is delayed there are other warps that can continue to execute as long as there is no serious contention on collector units. Our results in fact show that collector unit contention is very limited. However, the opportunities to hide the wake-up latency will diminish when there are not enough ready warps. For example, the number of warps running in nn,nw and sgemm benchmarks is less than 20. As a result, these benchmarks suffer the most performance degradation. On the other hands, the benchmarks with many active warps see no performance degradation.

Also, two benchmarks (heartwell and lbm) experienced performance improvement when the wake-up latency increased from one to two cycles. We analyzed the performance statistics for those benchmarks and it turns out that the additional wakeup delay lead to a different warp scheduling sequence. The newly scheduled warp sequence encountered fewer cache misses due to locality improvements. Hence the stalls due to the memory contention were reduced. As a result these benchmarks saw a slight performance improvement due to scheduling perturbations.

7 Related Work

Our work relies on three important prior works. Drowsy caches have been proposed as an efficient technique to reduce the leakage power consumption[12]. Also, [24] proposed a tri-modal switch that can be used to switch the combinational or sequential logic into one of three states, namely ON, off and drowsy. [17] and [28] discussed the DWL technique to avoid charging the wordline for the SRAM cells that are not part of the accessed word. In our study we modified these prior approaches and applied them in the context of GPGPU register file accesses.

CPU Register File and Cache Power Consumption : Dynamic and leakage power reduction techniques for CPU register file have been extensively studied. Techniques have been proposed to reduce the register file power consumption at the software level[7][26], the microarchitecture level[11] and the circuit level[16]. [14] used a compile time register file partitioning and code recompilation to reduce the number of active registers used. They divided the register file into active and inactive partitions. They used the drowsy technique to put the unused partitions in the drowsy mode. Their approach needs code recompilation and explicitly forces applications to use reduced register set to save power. Our approach neither needs code recompilation nor places any restrictions on register usage.

Cache leakage power reduction received significant attention. [12] and [18] proposed using the drowsy leakage current reduction technique on the data and instruction caches. [6] and [22] studied the leakage power reduction in caches. They used the prior knowledge of the cache access patterns and cache line inter-access time to apply the drowsy or cache line turn off techniques to reduce leakage power. When a cache line is turned off they rely on lower levels of memory hierarchy to fetch the data when needed. These techniques are applicable to cache designs in traditional CPUs but not for register files in GPGPUs. Turning off the register file after an extended idle period is not a viable option since there is no additional memory hierarchy levels to fetch the lost register data. In our work we studied the policies, the architecture and the circuit level modifications that are needed to reduce the GPGPU register file power consumption with negligible performance overhead.

GPU Power Consumption: GPU leakage power consumption reduction techniques have been proposed. In [27] the authors proposed saving the GPU leakage power through gating the unused processing elements. In [9] the working frequency and available SMs are dynamically modified during the application run to minimize the overall power consumption.

GPGPU Register file power consumption: In [13] the authors have studied the register file dynamic power reduction using a register file cache (RFC). RFC reduces the number of accesses to the main register file. RFC caches the registers that have been accessed recently by the active warps. This approach targets the dynamic power but not the leakage power. In [29] authors implemented the register file using the embedded DRAM(eDRAM). They divided the register file into a set of contexts. Each context holds the data for a set of warps. Every time they switch from one context to the other one they switch in the registers into the SRAM part of the memory and switch out the unused part into the DRAM part of the eDRAM. These two prior works on GPGPUs focused on reducing dynamic power of GPGPU register file. But in a Fermi-like configuration there are even greater opportunities to save on static power which our work exploits. Given the size of the register file in GPGPUs it is clear that static power reduction techniques will become critical going forward. Our work thus focuses on reducing the dynamic as well as static power consumption with negligible performance penalty and small hardware overhead.

8 Conclusion

GPGPU register file power consumption is a significant concern. Using a detailed workload characterization we show GPGPU-specific opportunities that can be exploited to reduce the register file power savings. In this paper we proposed two GPGPU-centric power saving techniques to reduce the static and dynamic power consumption of GPGPU register file. The first technique relies on a TRIC unit. TRIC uses a tri-modal switch to turn OFF registers that are not allocated for program execution. It then places all allocated registers into drowsy state by default and brings them to active state only when they are being accessed. Given the large inter-access distance to registers this aggressive drowsy state management mechanism suffers on average 1% performance overhead. The second technique relies on a COMA unit. COMA uses the active mask of a warp to eliminate the activation of the unused register segments in a wide register file organization. The two proposed techniques combined are able to reduce the total power consumption of the register file by 55% to 90%.

Acknowledgements: We would like to thank our paper shepherd Prof. Tor Aamodt for his valuable comments. This work was supported by DARPA-PERFECT-HR0011-12-2-0020 and NSF grants NSF-1219186, NSF-CAREER-0954211, NSF-0834798.

References

[1] Arizona state university predictive technology model. , http://ptm.asu.edu.

[2] Cacti 6.0: A tool to understand large caches. http://www.cs.utah.edu/ rajeev/cacti6/.

[3] Nvidia cuda sdk 4.2. developer.nvidia.com/cuda/cuda-downloads.

[4] Nvidia, fermi white paper v1.1. http://www.nvidia.com/content/PDF/fermi_white_papers/ NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.

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

[6] J. Abella, A. González, X. Vera, and M. F. P. O'Boyle. Iatac: a smart predictor to turn-off l2 cache lines. *ACM Transactions on Architecture and Code Optimiization*, 2(1):55–77, 2005.

[7] J. L. Ayala, A. Veidenbaum, and M. López-Vallejo. Power-aware compilation for register file energy reduction. *International Journal of Parallel Programming*, 31(6):451–467, Dec. 2003.

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

[9] J. M. Cebri'n, G. D. Guerrero, and J. M. Garcia. Energy efficiency analysis of gpus. In *IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW)*, pages 1014 –1022, May 2012.

[10] T.-J. Changhwan Shin, King, B. Liu, E. Nikolic, and Haller. Advanced mosfet designs and implications for sram scaling. *Technical Report*, 2011.

[11] J.-L. Cruz, A. González, M. Valero, and N. P. Topham. Multiple-banked register file architectures. In *Proceedings of the 27th annual international symposium on Computer architecture*, pages 316–325, 2000.

[12] K. Flautner, N. S. Kim, S. Martin, D. Blaauw, and T. Mudge. Drowsy caches: simple techniques for reducing leakage power. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 148 –157, 2002.

[13] 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 *Proceedings of the 38th annual international symposium on Computer architecture*, pages 235–246, 2011.

[14] X. Guan and Y. Fei. Register file partitioning and recompilation for register file power reduction. *ACM Transactions on Design Automation of Electronic Systems*, 15(3):24:1–24:30, May 2010.

[15] R. Ho. On-chip wires: Scaling and efficiency. *PhD Dissertation, Department of Electrical Engineering, Stanford University*, August 2003.

[16] J. Hu, T. Xu, and H. Li. A lower-power register file based on complementary pass-transistor adiabatic logic. *IEICE - Transactions on Information and Systems*, E88-D(7):1479–1485, July 2005.

[17] K. Itoh, K. Sasaki, and Y. Nakagome. Trends in low-power ram circuit technologies. *Proceedings of the IEEE*, 83(4):524 –543, April 1995.

[18] N. S. Kim, K. Flautner, D. Blaauw, and T. Mudge. Drowsy instruction caches: Leakage power reduction using dynamic voltage scaling and cache sub-bank prediction. In *Proceedings of 35th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 219 – 230, 2002.

[19] M. Kondo and H. Nakamura. A small, fast and low-power register file by bit-partitioning. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 40 – 49, February. 2005.

[20] X. Liang, K. Turgay, and D. Brooks. Architectural power models for sram and cam structures based on hybrid analytical/empirical techniques. In *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, pages 824 –830, November. 2007.

[21] M., A., M. Goodrum, J., A. Trotter, S. Aksel, T., K. Acton, and Skadron. Parallelization of particle filter algorithms. In *3rd Workshop on Emerging Applications and Many-core Architecture (EAMA)*, 2010.

[22] Y. Meng, T. Sherwood, and R. Kastner. On the limits of leakage power reduction in caches. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 154–165, 2005.

[23] 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 *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 308–317. ACM, 2011.

[24] E. Pakbaznia and M. Pedram. Design and application of multimodal power gating structures. In *International Symposium on Quality Electronic Design*, pages 120 –126, March 2009.

[25] S. Park, A. Shrivastava, N. Dutt, A. Nicolau, Y. Paek, and E. Earlie. Bypass aware instruction scheduling for register file power reduction. volume 41, pages 173–181, June 2006.

[26] S. Park, A. Shrivastava, N. Dutt, A. Nicolau, Y. Paek, and E. Earlie. Register file power reduction using bypass sensitive compiler. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(6):1155 –1159, June 2008.

[27] P.-H. Wang, C.-L. Yang, Y.-M. Chen, and Y.-J. Cheng. Power gating strategies on gpus. *ACM Transactions on Architecture and Code Optimimization*, 8(3):13:1–13:25, 2011.

[28] M. Yoshimoto, K. Anami, H. Shinohara, T. Yoshihara, H. Takagi, S. Nagao, S. Kayano, and T. Nakano. A divided word-line structure in the static ram and its application to a 64k full cmos ram. *IEEE Journal of Solid-State Circuits*, 18(5):479 –485, October. 1983.

[29] W.-k. S. Yu, R. Huang, S. Q. Xu, S.-E. Wang, E. Kan, and G. E. Suh. Sram-dram hybrid memory with applications to efficient register files in fine-grained multi-threading. In *Proceedings of the 38th annual International Symposium on Computer Architecture*, pages 247–258, 2011.