Buddy SM: Sharing Pipeline Front-End for Improved Energy Efficiency in GPGPUs

TAO ZHANG, Department of Computer Science and Engineering, Shanghai Jiao Tong University
NAIFENG JING, Department of Micro-Nano Electronics, Shanghai Jiao Tong University
KAIMING JIANG, WEI SHU, MIN-YOU WU, and XIAOYAO LIANG, Department of Computer Science and Engineering, Shanghai Jiao Tong University

A modern general-purpose graphics processing unit (GPGPU) usually consists of multiple streaming multiprocessors (SMs), each having a pipeline that incorporates a group of threads executing a common instruction flow. Although SMs are designed to work independently, we observe that they tend to exhibit very similar behavior for many workloads. If multiple SMs can be grouped and work in the lock-step manner, it is possible to save energy by sharing the front-end units among multiple SMs, including the instruction fetch, decode, and schedule components. However, such sharing brings architectural challenges and sometime causes performance degradation. In this article, we show our design, implementation, and evaluation for such an architecture, which we call *Buddy SM*. Specifically, multiple SMs can be opportunistically grouped into a buddy cluster. One SM becomes the master, and the rest become the slaves. The front-end unit of the master works actively for itself as well as for the slaves, whereas the front-end logics of the slaves are power gated. For efficient flow control and program correctness, the proposed architecture can identify unfavorable conditions and ungroup the buddy cluster when necessary. We analyze various techniques to improve the performance and energy efficiency of Buddy SM. Detailed experiments manifest that 37.2% front-end and 7.5% total GPU energy reduction can be achieved.

This article is extended from a six-page conference paper entitled "Dynamic Front-End Sharing in Graphics Processing Units," presented at the 32nd IEEE International Conference on Computer Design (ICCD). We do the further research and extend the conference paper in several directions:

—We propose an adaptive buddy cluster formation technique that can improve performance and save an average of 10% more energy than that in the conference paper (Sections 4.8 and 6.3).
—We construct simple large warp architectures and an eight-buddy architecture to compare with two-buddy and four-buddy architectures (Section 6.5). We evaluate additional regrouping strategies (Sections 4.6 and 6.2).
—We run additional experiments to evaluate the characteristics of the benchmarks and categorize them so that we can thoroughly analyze and explain the final performance results (Section 5.2).
—We perform new experiments to investigate the direct impact of the Buddy SM architecture on applications: issuing stalls and the memory access latency (Section 6.4).
—We add a background section to introduce the components in the SM front-end and the mechanism of instruction issue to help readers understanding of this work better (Section 3).

This work is partly supported by the National Natural Science Foundation of China (grants 61202026, 61332001, 61402285, and 61373155), the China Postdoctoral Science Foundation (grants 2013M540362 and 2014T70418), and the Program of China National 1000 Young Talent Plan.
Authors' addresses: T. Zhang, K. Jiang, W. Shu, M.-Y. Wu, and X. Liang (corresponding author), Department of Computer Science and Engineering, Shanghai Jiao Tong University, 800 Dong Chuan Road, Min Hang District, Shanghai 200240, China; emails: {tao.zhang, jkm_418, shu, mwu, liang-xy}@sjtu.edu.cn; N. Jing, Department of Micro-Nano Electronics, Shanghai Jiao Tong University, 800 Dong Chuan Road, Min Hang District, Shanghai 200240, China; email: sjtuj@sjtu.edu.cn.

DOI: http://dx.doi.org/10.1145/2744202

Categories and Subject Descriptors: C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors)—*Single-instruction-stream, multiple-data-stream processors (SIMDs)*

General Terms: Design, Experimentation, Performance

Additional Key Words and Phrases: GPU, front-end, energy efficiency

ACM Reference Format:
Tao Zhang, Naifeng Jing, Kaiming Jiang, Wei Shu, Min-You Wu, and Xiaoyao Liang. 2015. Buddy SM: Sharing pipeline front-end for improved energy efficiency in GPGPUs. ACM Trans. Architec. Code Optim. 12, 2, Article 16 (May 2015), 23 pages.
DOI: http://dx.doi.org/10.1145/2744202

1. INTRODUCTION

In recent years, general-purpose graphics processing units (GPGPUs) have experienced tremendous growth as general-purpose and high-throughput computing devices. GPGPU vendors keep adding architectural innovations that push the parallel processing capability of a GPGPU magnitudes higher than a CPU [NVIDIA Corporation 2012b; Brookwood 2010]. As a throughput-oriented device, a typical GPGPU chip has multiple computing engines called *streaming multiprocessors* (SMs), as in NVIDIA's terminology. The SMs are identical hardware clones that operate independently for the shader program execution. In this article, we define the logics in an SM for fetching, decoding, and issuing instructions as the pipeline front-end, which normally includes the first several pipeline stages, such as the instruction cache, the instruction buffer, the instruction decoder, the scoreboard, the warp scheduler, and the SIMT stack. Given the large number of concurrent thread warps (up to 48 in NVIDIA Fermi [NVIDIA Corporation 2009] architecture) in an SM for processing, the pipeline front-end usually takes a significant portion of the transistor budget and accounts for an average of 18% of the GPGPU's total dynamic power [Hong and Kim 2010; Lashgar et al. 2013]. Zhang et al. [2011] found that the fetch unit alone accounts for about 12% of the GPGPU's power and is the fourth most power hungry component. Figure 1 shows the power breakdown of the benchmarks from CUDA SDK 4.1 [NVIDIA Corporation 2012a], Parboil [Stratton et al. 2012], Rodinia [Che et al. 2009], and GPGPU-Sim 3.2.1 [Bakhoda et al. 2009] on a simulated NVIDIA GTX480 GPU.

In the GPGPU programming model (CUDA or OpenCL), kernels from an application are processed in a sequential order. Each kernel consists of multiple thread blocks, which can be thought as the instantiation of the kernels. A thread block can only reside in one SM, but it can be subdivided into one or more warps, with each warp encapsulating 32 threads. Different thread blocks are distributed across all the SMs in the round-robin manner. Thread blocks within the same kernel typically apply the same instruction stream but operate on different data. Current SM designs require every SM to operate independently, and the instruction execution flow is local to each individual SM. However, if thread blocks on different SMs come from a same kernel, the program instruction streams and the execution flows on different SMs will be quite similar or sometimes even identical for many GPGPU applications. This characteristic provides a unique opportunity for the resource sharing of the pipeline front-end units in different SMs, leaving room for power reduction in the fetch, decode, schedule, and other components.

In Zhang and Liang [2014], we proposed a preliminary architectural technique to share the GPGPU pipeline front-end units among SMs for improved energy efficiency. With this technique, neighboring SMs are grouped together into sharing clusters for synchronous execution. For example, a 16-SM GPU can be organized into four sharing clusters with four SMs in each cluster, as shown in Figure 2. SMs 1, 5, 9, and 13 become the masters in their corresponding clusters, whereas the rest become the slaves. All SMs in a sharing cluster proceed in a lock-step manner under the instrument of the

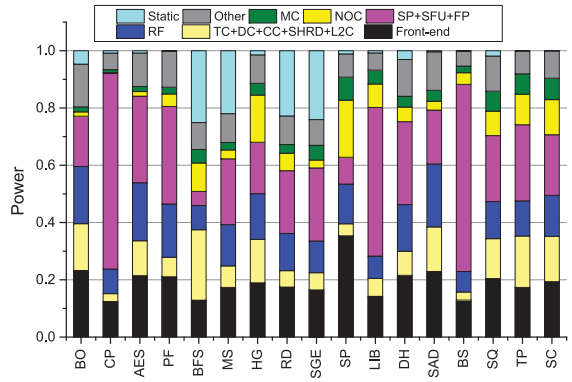

Fig. 1. Power breakdown of the benchmarks.

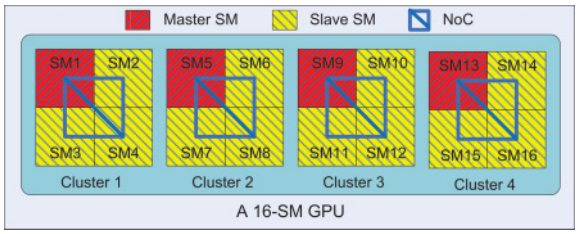

Fig. 2. An example for the buddy cluster splitting.

master. Different sharing clusters work independently without synchronization. The master's pipeline front-end is always powered on to work as usual, performing instruction fetching and decoding, scoreboard checking, and instruction issuing. However, the pipeline front-end units of all slaves are not in operation during the sharing period. Slave SMs simply accept commands from the master and follow the same operations. The execution, register file (RF), and memory units in the slaves are not affected and work as normal. This essentially means that all instructions executed in a sharing cluster are synchronized during the sharing period and the pipeline front-ends of all slaves can be powered off for energy savings.

To improve the preliminary technique in Zhang and Liang [2014], we do further study and extend the technique on several aspects. First, we propose a new adaptive buddy cluster formation technique to improve system performance. Second, we construct simple large warp architectures and an eight-buddy architecture to compare the performance with preferred two-buddy and four-buddy architectures. We also evaluate additional regrouping strategies. Third, we run additional experiments to evaluate the characteristics of the benchmarks and categorize them into regular and irregular and compute-intensive and memory-intensive applications so that we can thoroughly analyze and explain the final performance results. Fourth, we perform new experiments to investigate the direct impact of our front-end sharing architecture on applications: issuing stalls and the memory access latency. Finally, we add a background section to introduce the components in SM front-end and the mechanism of instruction issue to help readers for a better understanding of this work. We name the extended architectural technique *Buddy SM*.

In summary, this article makes the following major contributions:

—We propose a novel pipeline front-end sharing scheme called *Buddy SM*, which leverages the unique program execution behavior of thread blocks in GPGPUs for

energy savings. During the buddy period, only the front-end units in master SMs are working, whereas those of the slaves are powered off.
—We design and implement the Buddy SM scheme based on the NVIDIA Fermi architecture. We propose effective grouping, ungrouping, and regrouping mechanisms and an adaptive buddy cluster formation method to ensure the correct execution and good performance.
—We benchmark and categorize a suite of applications with diverse characteristics. Then we evaluate the Buddy SM scheme with these applications for different types of cluster formations and compare its performance with simple large warp architectures. We analyze the impact of the scheme on applications as well as the performance and energy efficiency results.

The rest of this article is organized as follows. Section 2 discusses the related work. Section 3 introduces the background. Section 4 presents the implementation detail of the Buddy SM architecture. Section 5 introduces the experiment methodology. Section 6 discusses the experiment results. Section 7 concludes the article.

2. RELATED WORK

Improving GPU energy efficiency has raised increasing attention in the architectural community in recent years. Lashgar et al. [2013] observed temporal instruction locality in GPU microarchitectures. They proposed to add a filter cache to eliminate 30% to 100% of instruction cache requests, which can reduce the front-end energy up to 19%. Their scheme also incurs hardware cost and cannot save power of many front-end units, unlike Buddy SM. There is a large body of works focusing on reducing power consumption of RFs in GPUs. Gebhart et al. [2011] observed that the registers are often read within three instructions after the write. They proposed an RF cache to reduce the number of accesses to the conventional power-hungry RF. They further introduced a two-level warp scheduler to prevent heavy contention on the cache. Several works have used emerging memory for RFs on GPUs to save energy. Jing et al. [2013] proposed using eDRAM as the replacement of SRAM for the on-chip storage including RFs to gain better energy efficiency. Goswami et al. [2013] integrated STT-RAM into GPUs as RFs. They introduced differential memory update, a merged register-write mechanism and a write-back buffer to coalesce multithreaded write accesses to save write energy. Yu et al. [2011] proposed using hybrid SRAM-DRAM RF architecture together with context switching mechanisms for energy savings. Differently from these works, we try to reduce GPU energy by sharing front-end units among several SMs.

In Zhang and Liang [2014], we proposed an architecture to opportunistically group and ungroup several neighboring SMs for execution. As far as we know, that work is the first to bind several SM processors to work in lock-step manner in GPUs. Combining several smaller cores into a single larger and more capable CPU core has been studied in previous works. Core fusion by Ipek et al. [2007] and core federation by Tarjan et al. [2008] were proposed, in which the cores of a homogeneous CMP were reconfigured at runtime into stronger cores by "fusing" resources from the available cores. However, they both use noncentralized hardware, and the scalability is limited by branch predictions, memory addresses, and the instruction window size. Another approach to the fusing of homogeneous cores is presented as composable lightweight processors [Kim et al. 2007], where 32 dual-issue cores could be fused into a single 64-issue processor. All of these schemes [Ipek et al. 2007; Tarjan et al. 2008; Kim et al. 2007] exhibit a high intercore communication overhead. Compared to these schemes, our scheme needs fewer architectural changes. The Voltron architecture from Zhong et al. [2007] allows multiple in-order VLIW cores of a chip multiprocessor (CMP) to combine into a larger VLIW core. The Cray X1 gangs together SSPs in an MSP for synchronous execution [Dunigan Jr. et al. 2005]. Our work does not try to combine

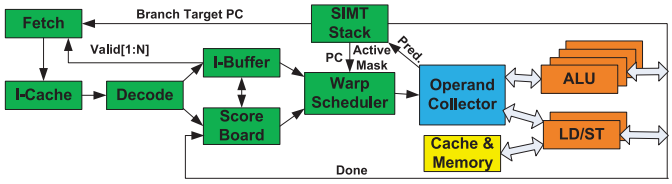

Fig. 3. Microarchitecture of an SM.

SMs to improve overall performance; instead, it attempts to power off most front-end units in GPUs and leave only one front-end in each buddy cluster to serve all of the back-ends to save energy.

To improve the resource utilization and performance of GPUs, various approaches have been proposed. Some studies tried to address the control divergences and the SIMD width effect issues. Fung et al. proposed the dynamic formation of warps [Fung et al. 2007] and later the thread block compaction [Fung and Aamodt 2011] to deal with diverging branches. Meng et al. [2010] proposed dynamic warp subdivision, which allows a single warp to occupy more than one slot in the scheduler to improve branch and memory divergence tolerances. Rhu and Erez [2012] introduced the CARPRI method for branch compaction, which avoids the unnecessary synchronization and only stalls threads that benefit from compaction. As a continuous work, the authors proposed the SIMD lane permutation [Rhu and Erez 2013b] to rearrange the mappings from threads to SIMD lanes for more effective thread compaction. The authors also proposed the dual-path execution model [Rhu and Erez 2013a] to allow the interleaved execution of two different paths. Narasiman et al. [2011] proposed the large warp microarchitecture and the two-level warp scheduling. Their large warp architecture creates fewer but larger warps that have a significantly larger number of threads than the SIMD width of the core, dynamically creating SIMD-sized sub-warps from the active threads in the large warps. Most of these works require the reorganization of threads every cycle and introduce hardware overhead to support such reorganization. Several works studied the impact of SIMD width on performance [Meng et al. 2012; Lashgar et al. 2012] and concluded that large warps improve the memory access coalescing but suffer more synchronization overhead, memory divergences, and branch divergences. Their evaluations showed that most benchmarks exhibited performance degradation when running large warps where NK-threaded warps running on K-wide SIMD lanes. Differently from these works, Buddy SM deals with branch divergences for correct execution instead of improved performance. Buddy SM tries to power-off as many front-end units as possible to save energy on diverse workloads, whereas the other works handle control divergence in irregular workloads for better performance.

3. BACKGROUND

In this section, we provide background information for the front-end units in GPGPUs, including the components and the working principles.

3.1. The Pipeline Front-End

A modern GPGPU consists of many small cores (the SMs). In the NVIDIA Fermi architecture [NVIDIA Corporation 2009], each SM has 32 data processing lanes with powerful arithmetic units, cache/shared memory, RFs, and a front-end unit, as shown in Figure 3. There are several major components in the front-end unit of an SM:

—*Instruction cache and buffer*: All active warps in an SM share an instruction cache. On the contrary, each active warp has its own instruction buffer. The instruction

buffer is used to buffer two to four decoded instructions. Every buffer entry has a valid bit to indicate if there is a nonissued instruction within this entry.

—*Instruction fetch and decode*: A warp is eligible for instruction fetch if it has no valid instructions left in its instruction buffer. All eligible warps are scheduled to access the instruction cache in the round-robin manner. Once selected, a read request is sent to the instruction cache with the address of the next instruction. If the instruction is found (instruction cache hit), the instruction buffer fetches it and sends it to the decoding stage directly. Otherwise, the instruction buffer sends out a memory request to fetch the instruction. Before the instruction is acquired from the main memory, the instruction buffer is held for later cache accesses. When the instruction shows up in the instruction cache, the instruction buffer will be arranged to fetch it again. At the decoding stage, the recent fetched instructions are decoded and stored in their corresponding buffer entries.

—*SIMT stack*: Each warp has its own SIMT stack to handle the control flow due to branch divergences. There are various techniques for reducing the negative impact of branch divergences. GPGPU-Sim 3.2.1 adopts the classic postdominator stack-based reconvergence mechanism introduced in Fung et al. [2009]. This technique tries to synchronize the divergent threads at the immediate postdominator to have more parallel threads running together after the reconvergence point. Each entry in a SIMT stack consists of a target PC, the immediate postdominator reconvergence PC, and the thread mask of the warp. At each divergent branch, a new entry is pushed onto the top of the stack. Later, the top-of-stack entry is popped when the warp reaches its reconvergence point. The SIMT stack is updated after every instruction issuance. The top-of-stack entry always records the exact next PC, the reconvergence PC, and the active mask that this warp should take, no matter whether a branch diverges or not.

—*Scoreboard*: The scoreboard in a GPGPU SM is similar to that in a CPU, which checks for WAR, WAW, and RAW dependency hazards. Each active warp has a corresponding entry in the scoreboard, and every instruction needs to perform a full scoreboard checking, causing power and area overhead. At the issue stage, the destination register of a warp instruction is recorded (register reserve) into that warp's scoreboard entry. Upon completion (writeback) of each instruction, the reserved register is removed (register release) from that scoreboard entry. Before the release of a register in the scoreboard, any dependent instruction that reads or writes to this register is held from issuing.

—*Schedule and issue*: All ready warp instructions that pass the scoreboard checking become candidates for issuing. In every clock cycle, the warp schedulers choose up to two instructions for issuing (so-called dual issue in Fermi), depending on the available issue slots. The choice of issuing warps depends on the scheduling policies, such as least recently issued, age based issue, priority based issue, or compiler directed issue policies. The scheduling policy has a significant impact on application performance. The warp schedulers can be power hungry depending on the implementation.

3.2. Instruction Issue Stage

At the issue stage, the warp scheduler checks all candidate warps for issuing. We use the greedy then oldest (GTO) scheduler [Rogers et al. 2012] in our experiments; this scheduler is the default scheduling scheme in GPGPU-sim. A warp is eligible to issue an instruction if it meets all the following conditions [Aamodt and Fung 2013]:

—The warp is not waiting at a barrier.
—It has valid instructions in its instruction buffer.
—An instruction passes the scoreboard checking.
—The instruction has no structural hazard.

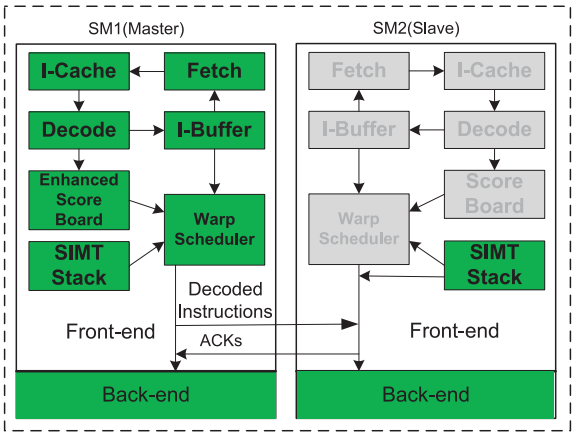

Fig. 4. Architecture of a Buddy SM cluster.

The structural hazard checking is performed by examining the availability of three pipeline status registers: ID_OC_SP, ID_OC_SFU, ID_OC_MEM. Instructions are issued to the corresponding function units based on their types. Memory instructions, including load/store and atomics, are issued to the memory pipelines. Other instructions are issued to SP or SFU units. The three status registers reflect the current availability status for different function units.

During the issue stage, if a branch divergence is detected, the instruction buffer of the warp is flushed and its next PC is updated to the target address from the SIMT stack. The detection is made by comparing the warp instruction's PC and the next PC in the top-of-stack entry of the SIMT stack. New instructions at the target address will be fetched and decoded, and this warp will become eligible for issuing again.

4. BUDDY SM SCHEME

4.1. Buddy Cluster Architecture

The basic Buddy SM scheme allows two or four neighboring SMs to form a *buddy cluster*. A GPGPU can be split into several buddy clusters, depending on its total number of SMs and the size configuration of the buddy cluster. Each buddy cluster contains a master SM and several slave SMs. SMs within a buddy cluster are called *cluster members*. There is a small network on chip (NoC) in the cluster for the communication between cluster members. The detail of the NoC will be presented in Section 4.7.

The architecture of an example two-buddy cluster is illustrated in Figure 4. SM 1 is the master, and SM 2 is the slave. In SM 1, all front-end components are activated. There is an enhanced scoreboard in SM 1 to track the data dependency for memory instructions (e.g., LD, Atomic) from all cluster members. For nonmemory instructions (e.g., ADD, Multiply), the scoreboard only checks the master's own data dependency. This is because nonmemory instructions have a same execution latency on the master and slaves, whereas the memory instructions vary in latency on different SMs. The warp schedulers of the master determine the instructions to be issued to the entire cluster. All SMs in a cluster work in a synchronous manner.

In the slave SM 2, all front-end components are power gated except for the SIMT stack. Slave SMs still manage their own SIMT stacks for recording branch divergences and reconvergence conditions. This is useful when the cluster is ungrouped, after which every SM in the cluster will operate independently. Details will be presented in Section 4.5.

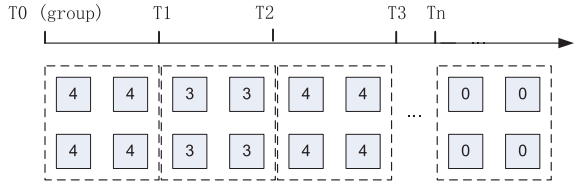

Fig. 5. Uniform CTA dispatching example.

4.2. Instruction Issue

One key question is how master SMs schedule and issue instructions. In every issue cycle, each master SM checks the criteria presented in Section 3.2 to decide if a warp instruction can be issued. In most cases, it only needs to check its local information (SIMT stack, scoreboard, execution units status, etc.) because for the fully synchronized execution in the buddy cluster, the data path in all SMs will exhibit exactly the same behavior. To issue an instruction to the slaves, the master needs to send decoded instructions through the NoC in the cluster.

However, there are exceptions for memory-related operations. Since the latency for accessing memory can be different across SMs, we use an enhanced scoreboard in each master to track the completion of memory accesses. An enhanced scoreboard is implemented by adding four more monitoring bits to each scoreboard entry, as there are at most four SMs in each cluster (in the four-buddy configuration). Slaves are required to send "ACKs" to their master through the NoC to acknowledge the completion of memory accesses including the shared and global memories. A corresponding monitoring bit is cleared when a master receives an ACK or the master itself completes a memory access. When all monitoring bits of a scoreboard entry are cleared, this entry will be removed from the enhanced scoreboard. Nonmemory instructions do not need acknowledgments because of the fixed latency.

4.3. Uniform Concurrent Thread Array Blocks Dispatching

Each GPU has a global concurrent thread array (CTA) scheduler for allocating thread blocks onto SMs. In conventional design, SMs work independently and can be assigned a different number of thread blocks. However, to maintain the correct Buddy SM operation, the CTA scheduler needs to assign an equal number of blocks for all SMs in a cluster. Otherwise, either the master may issue instructions not existing on slaves or some instructions on slaves never get issued. Either case will cause error. In the Buddy SM scheme, every SM in a cluster will receive an equal number of thread blocks at the beginning of execution. When all SMs in the cluster finish a same thread block, the CTA scheduler will dispatch a new set of blocks onto the SMs. A K-bit vector is used to track each buddy cluster where K is the buddy size. An SM set its bit in the vector when it can receive a new thread block. The CTA scheduler can dispatch thread blocks to SMs in a cluster when all bits are set in the clusters vector.

Figure 5 illustrates a scenario in a four-buddy cluster. At time $T0$, each SM in this cluster receives four blocks and begins to execute. At $T1$, all SMs finish one block and are assigned a new block at $T2$. At Tn, all of the blocks are finished. However, a GPU application may have multiple kernels with various number of blocks. The number of blocks may not always be dividable for the buddy size. This might cause performance issues for Buddy SM, as described in detail in Section 4.8.

4.4. Grouping

Normally, a GPU application consists of multiple kernels. The Buddy SM scheme will split a GPU into clusters at the time to launch a new kernel onto the GPU. Depending

```
__global__ void
Kernel( Node* vertices, int* edges, bool* mask, bool*
      updating, bool *visited, int* g_cost, int nVertices)
{
    int tid = blockIdx.x*blockDim.x + threadIdx.x;
    if( tid<nVertices && mask[tid]) {
        int  base= vertices[tid].starting;
        mask[tid]=false;

        for(int i=base; i< base + vertices[tid].nEdges; i++) {
            ...; //code to visit edge i
        }
    }
}
```

Fig. 6. Pseudocode of the BFS kernel.

on the buddy size N, all N adjacent SMs are grouped into a buddy cluster. In each cluster, the SM with the minimum index will become the master, and the rest of the SMs become the slaves. For example, $SM_i, i = 2k+1, k = 0, 1, 2 \ldots$ will become masters in the two-buddy case, and $SM_i, i = 4k + 1, k = 0, 1, 2 \ldots$ will become masters in the four-buddy case. With this design, at most half of the SMs in a GPU can be masters and need to implement the enhanced scoreboard in hardware. The SMs within each cluster will be in synchronous execution. Clusters work independently with each other.

4.5. Ungrouping

The state-of-art GPGPUs utilize thread masks to distinguish branch directions. At every branch, threads in a warp going in one direction will set their bits in the thread mask to "1," whereas the others will set their bits in the mask to "0." In Buddy SM, the prerequisite for coupled execution is that the master and slaves in a buddy cluster execute the same instruction flow. Nevertheless, this is not always true even if SMs are processing thread blocks from the same kernel. When SMs in a buddy cluster execute different paths (called *SM divergence*), the cluster will be ungrouped. The SM divergence occurs more frequently in irregular applications. Figure 6 shows the pseudocode of the BFS kernel [Bakhoda et al. 2009], demonstrating how the SM divergence occurs in a four-buddy cluster. As shown in Figure 6, each GPU thread processes a vertex in a graph, iterating on its edges and looking for unvisited one-hop neighboring vertices. However, vertices may have different numbers of edges, and thus different threads may finish the *for* sentence at different times. As a result, SMs executing the warps and blocks will differ in their thread masks.

We evaluated a wide range of applications, and most of them have no SM divergence. For all other cases, such as when warps on SMs traverse all branch directions or take the same direction, this is not divergence and thus Buddy SM will not be affected. Besides, in almost all cases when SM divergence occurs, buddy master and slaves will have different thread masks that can easily be identified. In very rare cases, the SM divergence happens even if SMs have the same thread masks. This is caused by the involvement of the thread ID into a condition evaluation so that the branch target addresses differ. To make our scheme simple and efficient, we leave this uncommon case to the compiler. A compiler can easily identify such cases and signal the hardware using the compiler hint. In the Buddy SM execution, after a branch instruction has been executed (thus the thread masks are determined or the compiler hint is resolved), the master will broadcast its thread masks to all cluster members. All slaves will compare their own thread masks with those of the master. If they diverge, the slaves will send an "ungroup" request to the master. The master will ungroup the cluster

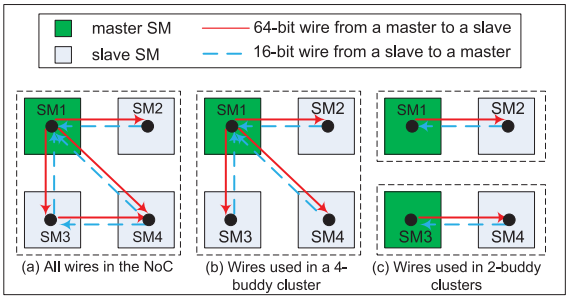

Fig. 7. NoC in a buddy cluster.

when it receives an ungroup request packet from any of the slaves. After a ramp-down period, all SMs will run independently without the front-end sharing. Since the slaves keep their own SIMT stacks as shown in Figure 4, a slave will follow its own branch direction indicated in its local SIMT stack after ungrouping.

There are couple of things happening during the ramp-down period. First, the front-end units of the slaves have to be powered up, taking dozen of clock cycles. Second, since the slaves do not have scoreboard information (in the master only), they have to wait for the items in the master scoreboard to be completely cleared before they can start executing on their own. The waiting time of the preceding two usually causes overlaps. Third, the slaves need some warmup time for the instruction cache. The ungroup/ramp-down takes place maximally once in every kernel; the new kernel regrouping strategy is presented in the next section. When executing thread blocks from a kernel, a buddy cluster is ungrouped when it encounters the first SM divergence. The costs of grouping and ungrouping are included in performance results.

4.6. Regrouping

After ungrouping, the clusters will have chances to regroup again. We consider two regrouping strategies: new kernel and halfway. Normally, GPU applications consist of multiple kernels, with each implementing a certain function. Under the new kernel strategy, buddy clusters, once ungrouped, will not try to regroup in current kernel. At the beginning of the next new kernel, SMs will be arranged to regroup for the Buddy SM execution even if they are only ungrouped in the last kernel.

The halfway regrouping strategy is an alternative way to regroup the ungrouped clusters at the next common reconvergence point. With this strategy, ungrouped clusters have more chances to regroup for the Buddy SM execution. However, SMs may reach the reconvergence point at different times. One SM may get to the reconvergence point earlier than other SMs, resulting in extra stalls. In addition, programs with this strategy may ungroup and regroup multiple times in a kernel, causing more overhead due to ungrouping operations during the ramp-down period. We will evaluate the performance of these two strategies for comparison.

4.7. Communications in Clusters

In Buddy SM, each buddy cluster has a NoC for the communication between the master and slaves. Figure 7(a) shows the connections of the NoC. There is a pair of wires connecting the master and each slave. The downlink connection from a master to a slave is 64 bits wide carrying the packets of decoded instructions. The uplink connection from a slave to a master is 16 bits wide carrying the acknowledgments and other information. Figure 7(b) shows the wire connections for a four-buddy cluster, and Figure 7(c) shows the connections for two two-buddy clusters. Because of the

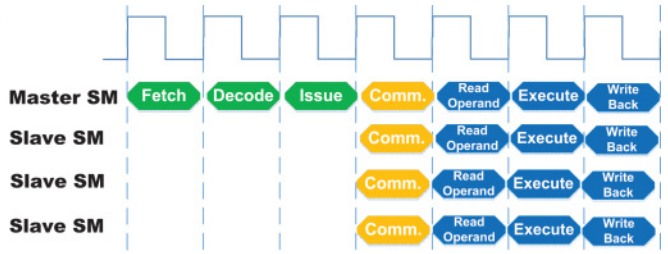

Fig. 8. Timing of pipeline stages.

fixed master design explained in Section 4.4, the SM with the minimum index in each cluster will become the master. For example, SM 1 becomes the master in the four-buddy configuration, whereas SM 1 and SM 3 will be the masters in the two-buddy configuration. Same as the GPU interconnection network between the SMs and the L2 cache, the buddy cluster NoC operates at twice the frequency of the SM core. However, the buddy cluster NoC is totally 10 bytes (64 bit + 16 bit) wide, whereas the width of the GPU interconnection network is 32 bytes.

There are three major types of packets on the NoC: *InstPacket* contains decoded instruction information; *MemPacket* contains memory access ACK messages; and *CtrlPacket* controls the cluster behavior, such as ungrouping. CtrlPackets contribute a negligible portion of the total packets. Depending on the memory intensiveness, MemPackets can take a significant portion of the network traffic. The following present the detailed information for each type of packet:

—*InstPacket*: This packet has 64 bits. The first 32 bits include a 6-bit warp ID, a 4-bit function unit ID, a 6-bit operation ID, and a 16-bit immediate number. The second 32 bits contain five 5-bit register IDs. If the instruction is a branch, there is an additional packet containing the master's thread mask.
—*MemPacket*: This packet has 16 bits. The slaves will acknowledge for the completion of each memory access with a packet. An ACK packet has a 2-bit access type, a 3-bit slave ID, a 6-bit warp ID, and a 5-bit register ID.
—*CtrlPacket*: Currently, there is only an "ungrouping" message sent in this type of packet. The packet has a 1-bit type ID and a 3-bit slave ID. The type ID is set to 1 for ungrouping.

Each Fermi SM has two warp schedulers. Therefore, at most two instructions can be issued in each SM core cycle. Since the NoC operates at twice the frequency of the core, they have enough bandwidth to transfer two instructions at the core cycle. Figure 8 shows the timing of the pipeline stages for a four-SM cluster. Besides the regular pipeline stages, a new "communicate" stage is inserted between the issue and the read operand stage. The instruction transfer from a master to its slaves should be made in this stage. As shown in Figure 9, the Fermi GPU chip has roughly $23mm \times 23mm$ die size and is manufactured in 40nm technology [Valich 2010], so we estimate that the communication stage takes one-cycle latency to traverse the distance between two adjacent SMs, which is roughly 5mm long. The delay of 5mm wire is 0.3ns (3.3GHz) reported by CACTI 6.0 [Muralimanohar et al. 2009]. We clock the NoC at 1.4GHz, which not only meets one cycle latency but also provides headroom for low-swing voltage operation that causes 0.6ns wire delay and 0.1ns signal regeneration that still meets timing. Low swing for wired bus is a common technology when latency is not critical. CACTI 6.0 reports 50fJ/bit/mm for low swing, about one sixth the energy of the full swing. Transferring a 64-bit instruction requires 50*64*5=16pJ, much less than reading a 64-bit instruction from a 4KB I-cache incurring 32pJ by CACTI 6.0.

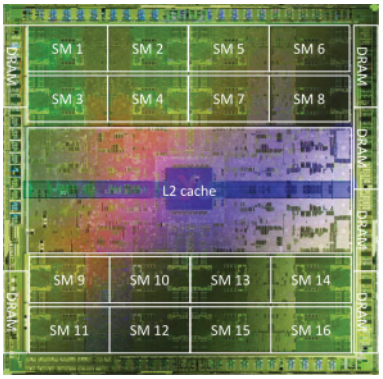

Fig. 9. Die photo of a GTX480 GPU.

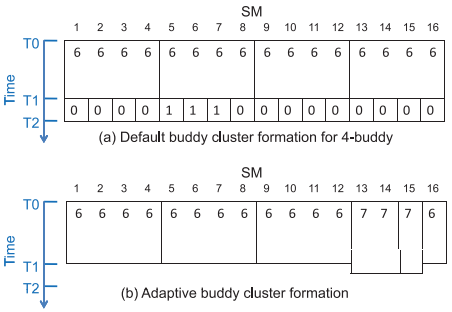

Fig. 10. Adaptive buddy cluster formation example.

Furthermore, the slaves power off I-caches, saving leakage power. The I-cache miss power also reduces because slaves never fetch instructions from main memory. The total area of the cluster NoC is estimated to be 2.3% of the area of the GPU interconnection network between SMs and the L2 cache. The estimation is calculated based on the number of links (wires), the width of the links, and the length of these links.

4.8. Adaptive Buddy Cluster Formation

The default buddy cluster formation scheme will form M/N clusters with N SMs in each cluster for totally M SMs in a GPU. As shown in Figure 10(a), a 16-SM GPU is split into four clusters with four SMs in each cluster. However, this default buddy cluster formation might cause a performance issue under the uniform CTA dispatching rule described in Section 4.3.

The number of thread blocks (denoted as B) in a GPU kernel is determined by the amount of input data and the algorithm. Therefore, it might not always be dividable by N. For example, the sum of absolute difference (SAD) [Stratton et al. 2012] has a kernel of 99 thread blocks. With the uniform CTA dispatch rule, each SM will receive six thread blocks from the first wave of thread block dispatch at time $T0$, as shown in Figure 10(a). There are three remaining blocks that cannot be accepted by any cluster because the cluster has to receive four blocks at a time. Since there are remaining blocks, one cluster will be ungrouped to process the last three blocks at time $T1$ when it completes all dispatched thread blocks. This causes starvation on other SMs and the overall extended execution time.

We propose an adaptive buddy cluster formation method to solve the issue. The general idea is to split one cluster at the very beginning of the thread block dispatching according to the reminder of B/N. The adaptive buddy cluster formation for a 16-SM GPU is shown in Table I. SMs of the last split cluster will process the residue blocks. For example, for four-buddy and three residue blocks, the last cluster is split into one two-buddy cluster and two independent SMs. All blocks can be distributed at time $T0$, where SM 13, 14, and 15 accept one more block, as shown in Figure 10(b). We call the SMs that handle the residue blocks *responsible* SMs. Other cases are shown in Table I. The adaptive buddy cluster formation process is made at the launching of each new kernel when the basic kernel information including block count is available. This method speeds up the execution and greatly reduces the performance overhead in Buddy SM.

Table I. Adaptive Buddy Cluster Formation

Buddy Size	Residue Blocks (#)	Cluster Formation	Responsible SM(s)
Four buddy	1	4,4,4,2,1,1	15
Four buddy	2	4,4,4,2,2	13,14
Four buddy	3	4,4,4,2,1,1	13,14,15
Two buddy	1	2,2,2,2,2,2,1,1	15

5. EXPERIMENT METHODOLOGY

5.1. Power and Area Cost

5.1.1. Buddy Cluster NoC. We model the power of the buddy cluster NoC using GPUWattch [Leng et al. 2013]. GPUWattch is an integrated power modeling tool in the GPGPU-Sim simulator [Bakhoda et al. 2009] that can report the runtime power of each component. The Fermi architecture that we simulated has a crossbar interconnection network for the communication between SMs and the L2 cache. Figure 9 shows the die photo [Sandhu 2010] of a GTX480 GPU. We marked the regions of each SM, the L2 cache, and the DRAM controllers as indicated by NVIDIA Corporation [2009]. We can see that the middle part of the chip is the L2 cache with the cache controller at the center. At either side of the L2 cache, there are eight SMs closely fitted. Two and four memory controllers are located on the left and right sides of the chip, respectively.

We operate the buddy cluster NoC at 1.4GHz, as in the GPU interconnection network, which is twice the 700MHz SM core frequency. However, the buddy cluster NoC and the GPU interconnection network differ in the number of links, average packet traverse distance, NoC width, and amount of data transferred. We record the amount of data transferred for both the buddy cluster NoC and the GPU interconnection network at runtime. We also measure the average travel distance in Figure 9. We assume that the power of the interconnection network and the buddy cluster NoC is linearly proportional to the average distance and amount of data transferred. Since GPUWattch can report the power consumption of the interconnection network, we can calculate the power of the buddy cluster NoC.

5.1.2. Enhanced Scoreboard. Four bits need to be added for each entry in a master SM's scoreboard. Each scoreboard has at most 48 entries, as the prototype Fermi GPU architecture supports a maximum of 48 active warps per SM. Because of the fixed master design elaborated in Section 4.4, only half or a quarter of the SMs in a GPU can become masters in two-buddy and four-buddy configurations, respectively. Therefore, it accounts for 1,536 bits and 768 bits to enhance all scoreboards in a 16-SM GPU under two-buddy and four-buddy configurations, respectively.

One-hot coding is used in the added four bits to track the memory operations of the master and at most three slaves. We linearly scale the scoreboard power according to the ratio of the added bits over the original bits. This is because the scoreboard is mostly performing matching operations. One bit stands for one set of matching logics, and the total power is proportional to the number of bits in the scoreboard. Because of the fixed master design, at most half of the SMs in the GPU will incur this power overhead. The area of these extended bits is quite small and can be ignored. There is a trade-off between area and modular SM design and verification. To save area, one can implement the master specific addons only on the master SMs. However, for modular SM design and verification, these addons can be implemented on all SMs.

5.2. Benchmarks

Our simulated workloads are summarized in Table II. To show the generality of the proposed schemes, we mix benchmarks from four sources: NVIDIA CUDA SDK 4.1

Table II. Simulated Workloads

Name	Abb.	gridDim	blockDim	Instructions
Binomial options [NVIDIA Corporation 2012a]	BO	(64,1,1)	(256,1,1)	1406.34M
Coul. potential [Bakhoda et al. 2009]	CP	(8,32,1)	(16,8,1)	120.13M
AES encryption [Bakhoda et al. 2009]	AES	(257,1,1)	(256,1,1)	28.67M
PATH finder [Che et al. 2009]	PF	5*(463,1,1)	(256,1,1)	619.16M
BFS search [Bakhoda et al. 2009]	BFS	8*(16,1,1)	(256,1,1)	0.88M
Merge sort [NVIDIA Corporation 2012a]	MS	(32,1,1)	(512,1,1)	62.50M
		15*(1,1,1)	(256,1,1)	
		5*(256,1,1)	(128,1,1)	
Histogram [NVIDIA Corporation 2012a]	HG	2*(18,1,1)	(64,1,1)	34.33M
		2*(239,1,1)	(192,1,1)	
		2*(256,1,1)	(256,1,1)	
		2*(64,1,1)	(256,1,1)	
Reduction [NVIDIA Corporation 2012a]	RD	100*(1,1,1)	(16,1,1)	38.46M
		101*(32,1,1)	(256,1,1)	
Sgemm [Stratton et al. 2012]	SGE	(1,31,1)	(16,8,1)	8.87M
Scalar prod [NVIDIA Corporation 2012a]	SP	4*(128,1,1)	(256,1,1)	48.06M
Swap. portfolio [Bakhoda et al. 2009]	LIB	2*(64,1,1)	(64,1,1)	952.84M
DwtHaar1D [NVIDIA Corporation 2012a]	DH	(256,1,1)	(512,1,1)	18.89M
		(1,1,1)	(128,1,1)	
Sum of absolute difference [Stratton et al. 2012]	SAD	(11,9,1)	(32,1,1)	286.16M
		(11,9,1)	(32,4,1)	
		(44,36,1)	(61,1,1)	
Black scholes [NVIDIA Corporation 2012a]	BS	(480,1,1)	(128,1,1)	371.72M
Sobol QRNG [NVIDIA Corporation 2012a]	SQ	(1,100,1)	(64,1,1)	210.47M
Transpose [NVIDIA Corporation 2012a]	TP	16*(64,64,1)	(16,16,1)	724M
Scan [NVIDIA Corporation 2012a]	SC	9*(6656,1,1)	(256,1,1)	1348.95M

[NVIDIA Corporation 2012a], Parboil [Stratton et al. 2012], Rodinia [Che et al. 2009], and the benchmark suite from GPGPU-Sim [Bakhoda et al. 2009]. We select the applications for diversity: there are memory-intensive applications and compute-intensive applications, as well as regular and irregular applications. In the table, the "gridDim" describes the number of kernels and thread blocks. For example, the ScalarProd has four kernels of the dimension (128,1,1). The "blockDim" refers to the dimension of threads in a thread block. From the table, we can see that most applications have multiple kernels, and the beginning of each kernel can be the regrouping point of Buddy SM.

We carry out experiments with GPGPU-Sim to characterize these benchmarks. We use the default settings in the simulator without Buddy SM and execute applications one by one with different numbers of SMs to study their performance and memory access latency. Figure 11 shows their performance results normalized to that of a single SM. From the figure, we can conclude that BinomialOptions (BO), Coul. Potential (CP), AES, and PATH Finder (PF) are compute-intensive applications whose performance keeps increasing with the growing number of SMs. On the contrary, the performance of some other applications, such as BFS, varies very little with the number of SMs, possibly limited by the memory bandwidth or the amount of parallelism.

Figure 12 presents the average memory access latency normalized to that of a single SM. Since the entire memory capacity and bandwidth are fixed, increasing SMs, and consequently more concurrent threads, will introduce more contentions on memory accesses. From the figure, we can see that BlackScholes (BS), SobolQRNG (SQ), Transpose (TP), and Scan (SC) are memory-intensive applications, as their average memory access latency increases significantly with the growing number of SMs but their performance changes only a little.

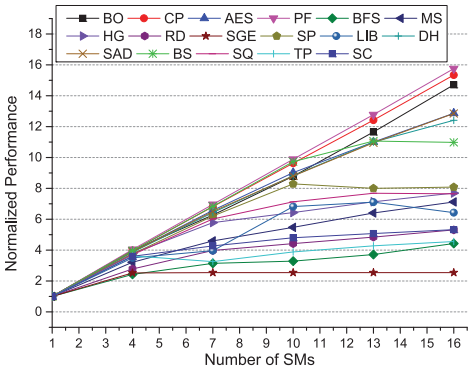

Fig. 11. Normalized performance.

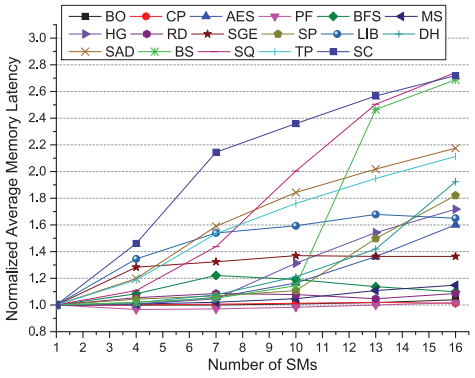

Fig. 12. Normalized average memory latency.

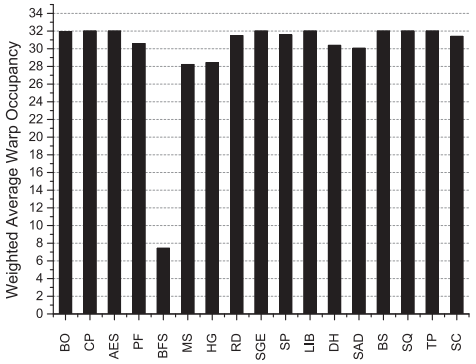

Fig. 13. Weighted average warp occupancy.

Figure 13 shows the weighted average warp occupancy [Burtscher et al. 2012] for all cycles where an instruction is issued to the pipeline based on the active masks for all warps at the issue stage in each scheduler. The warp size is 32, but the 32 threads are not active all the time due to branch divergences. A program free of control-flow divergences can reach a weighted average warp occupancy of 32. Such applications are considered as regular workloads that do not suffer from thread divergences. On the contrary, an irregular application will have a relatively smaller weighted average warp occupancy [Burtscher et al. 2012]. From the figure, we can see that BFS, Merge-Sort (MS), and Histogram (HG) are irregular applications whose runtime behavior is determined by the input data. Both the control flow and the memory access patterns may vary depending on the input data. Irregular codes usually arise from the use of dynamic data structures, such as trees and graphs.

For the analysis, we conclude that our selection of benchmarks covers both compute-intensive and memory-intensive workloads, as well as regular and irregular workloads. Therefore, these applications are suitable for the evaluation of the Buddy SM scheme.

5.3. Simulator Configurations

We use GPGPU-Sim 3.2.1 [Bakhoda et al. 2009] as our simulation platform. We adopt the settings for an NVIDIA GTX480 GPU (Fermi architecture). The machine parameters are listed in Table III. The default GTO scheduler [Rogers et al. 2012] is used as the warp scheduler. We evaluate the GPU and workload behavior under the two-buddy,

Table III. Simulator Architectural Configuration

Configuration Items	Value
Shaders (SMs)	16
Warp size	32
Capacity/core	MAX. 1,536 threads, 48 warps, 8 CTAs
Core/memory clock	700MHz/924MHz
Interconnection network	1.4GHz, 32 bytes wide, crossbar
Registers/core	32,768
Shared memory/core	48KB
Constant cache/core	8KB, two-way, 64B line
Texture cache/core	4KB, four-way, 128B line
L1 data cache/core	32KB, four-way, 128B line
L1 I-cache/core	4KB, four-way, 128B line
L2 cache	64KB, 16-way, 128B line
Warp scheduler	Greedy then oldest (GTO)
DRAM model	FR-FCFS memory scheduler, 6 memory modules

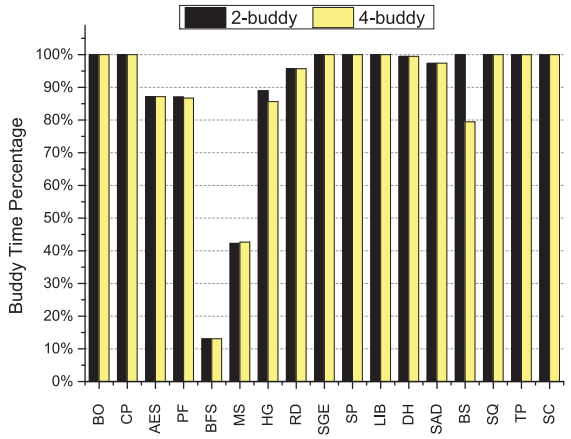

Fig. 14. Buddy time percentage.

four-buddy, and eight-buddy configurations. Runtime statistics including performance and power numbers are captured for each benchmark for analysis. The power consumption of each component is acquired from GPUWattch [Leng et al. 2013] inside GPGPU-Sim.

6. RESULTS AND ANALYSIS

This section presents the experiment results and analyzes the insights. The results include the buddy time percentage, evaluation of the regrouping strategies and the adaptive buddy cluster formation, impact of Buddy SM, performance, and energy savings of Buddy SM.

6.1. Percentage of Buddy Time

Figure 14 shows the average percentage of cycles in the Buddy SM mode in total execution cycles (called *buddy time percentage*). A GPU application often has multiple kernels, and the buddy time percentages of these kernels can be different. The results shown in this figure are the average value of all kernels of each application. A 100% means that SMs in all buddy clusters are in the Buddy SM execution during an application's entire lifetime. For Buddy SM, divergence in a cluster means that a warp

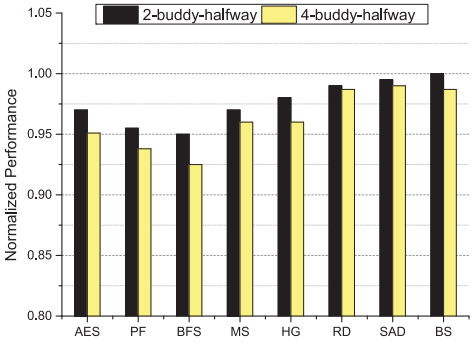

Fig. 15. Evaluation of the regrouping strategies.

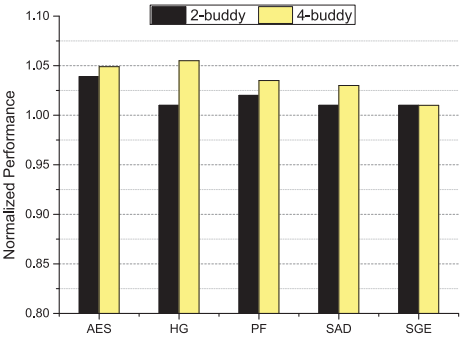

Fig. 16. Evaluation of the adaptive buddy cluster formation.

on an SM takes one direction and a warp on another SM takes another direction in a branch, under which Buddy SM will ungroup. For all other cases, such as when the warps on SMs traverse all branch directions or take the same direction, this is not divergence and thus Buddy SM will not ungroup. This is why many applications in the figure have 100% buddy time percentage. Irregular applications, such as BFS, MS, and HG, all have a buddy time percentage less than 100%. In general, the percentage is determined by how often and where the SMs diverge. Applications that have frequent SM divergences, or in which the SM diverges at an early stage of the kernels, will have smaller buddy time percentage.

6.2. Evaluation of the Regrouping Strategy

Figure 15 shows the performance comparison of two regrouping strategies. The performance of the Buddy SM execution with the halfway regrouping strategy is normalized to the performance with the new kernel regrouping strategy. The regrouping strategies only affect the benchmarks that have SM divergence. Therefore, Figure 15 only shows the benchmarks whose buddy time percentage is less than 100%. The new kernel regrouping strategy outperforms the halfway strategy. On average, the halfway strategy achieves 97.6% and 96.2% performance of the default new kernel regrouping strategy under two-buddy and four-buddy configurations, respectively. Benchmarks obtain lower performance under four-buddy configurations since more waiting cycles are incurred due to more members in each buddy cluster. Two benchmarks, PF and BFS, achieve the lowest performance since they experience more times of ungrouping and regrouping in kernels than other benchmarks, wasting many cycles on the ramping down operations (see Section 4.5) and so on. Although the halfway regrouping strategy can improve the buddy time percentage of some benchmarks, it cannot save more energy than the default new kernel regrouping strategy, according to the experiment results. This is because the performance degradation of the halfway strategy results in prolonged execution time of the entire GPU, consuming more energy than it saves. Therefore, the new kernel regrouping strategy is used in all later experiments.

6.3. Evaluation of the Adaptive Buddy Cluster Formation

Figure 16 shows the performance of the adaptive buddy cluster formation method under the two-buddy and four-buddy configurations. Five applications have kernels with a nondividable number of thread blocks: AES, HG, PF, SAD, and SGE. The performance shown in the figure is normalized to the performance with the default uniform block dispatching scheme. We observe improved performance on all five benchmarks. For

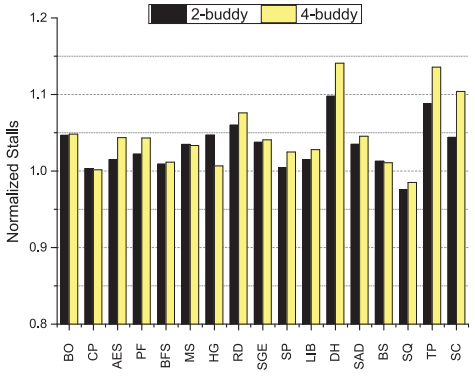

Fig. 17. Average issue stalls.

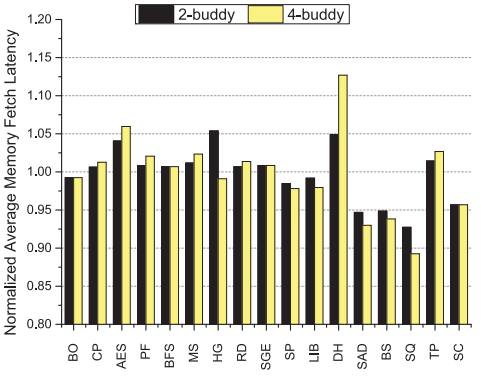

Fig. 18. Average memory fetch latency.

all other applications not shown in this figure, the adaptive buddy cluster formation method has the same performance with the default scheme. Therefore, we always adopt the adaptive buddy cluster formation method in the following experiments.

6.4. Impact of Buddy SM

Buddy SM employs synchronized execution on SMs in a buddy cluster and therefore might affect the instruction issue and memory accesses compared to the conventional scheme. Applications can possibly experience a slight performance degradation due to the influence. Buddy SM imposes additional constraints on the instruction issue—the resource requirements on all cluster members must be met before a dependent instruction can be issued. However, there are multiple active warps (48 active warps for Fermi) on each SM, so this does not necessarily cause a problem as long as the schedulers can issue instructions from other active warps. Stalls may occur if none of the active warps is ready.

Figure 17 shows the total number of stalls under the two-buddy and four-buddy configurations, respectively. The results are normalized to that in the normal GPGPU execution without Buddy SM. Most applications only suffer a minor increase of stalls. Besides, the number of stalls generally increases with the buddy size because of the more stringent requirement for cluster synchronization. We do observe the anti-trend that some applications receive fewer stalls, such as HG and SQ. A further look reveals that the instruction issuing order has been changed by Buddy SM, which in some cases reduces the stalls.

Besides the stalls for the instruction issue, the memory access latency can also be affected by Buddy SM. In the traditional scheme, all SMs proceed independently so that their memory access patterns are relatively random. However, with Buddy SM, which requires synchronized execution, the memory accesses from SMs in a buddy cluster become more aligned. The change of the memory access pattern sometimes causes congestion on the crossbar and increases memory access latency. Note that buddy clusters operate independently so that the memory accesses from different clusters are still distributed over time. On the other hand, aligned memory accesses create more opportunities for memory coalescing and Buddy SM reduces instruction fetching bandwidth, which might be beneficial to the memory latency. Depending on the applications' original memory access patterns and intensities, the overall memory access latency under Buddy SM can become worse or better. Figure 18 shows the average memory fetch latency under two-buddy and four-buddy configurations, respectively. The results are normalized to that in the normal GPGPU execution. The latency increases for half of

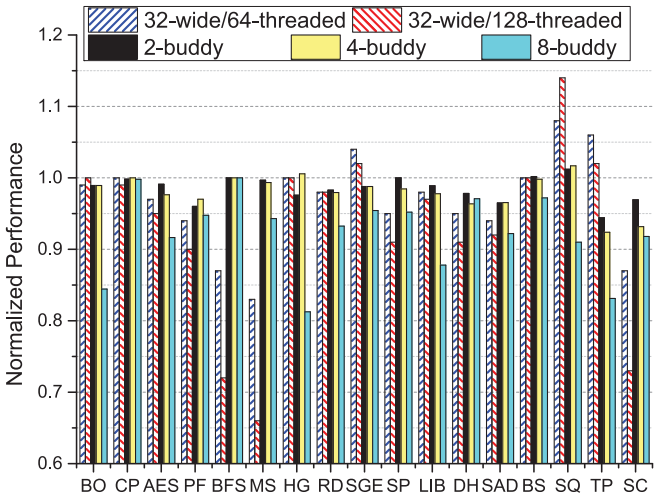

Fig. 19. Normalized performance.

the applications and decreases for the rest. DH has a significant increase in its latency, whereas SQ shows an obvious reduction. In general, applications with an increased number of stalls and memory fetch latency will exhibit performance degradation.

6.5. Performance

Figure 19 compares the performance in instructions per cycle (IPC) of five architectures: 64-threaded warp, 128-threaded warp, two-buddy, four-buddy, and eight-buddy. All SMs have 32-SIMD lanes. A 64-threaded warp and a 128-threaded warp take two and four cycles to execute on 32-SIMD lanes, respectively. The results are normalized to the performance of the default (baseline) 32-threaded warp architecture.

On average, five architectures achieve 96.3%, 92.7%, 98.5%, 97.9%, and 91.3% normalized performance for all benchmarks, respectively. Two-buddy and four-buddy configurations achieve better performance than their corresponding large warp architecture. As analyzed in Meng et al. [2012] and Lashgar et al. [2012], large warps can improve the memory access coalescing but will suffer more synchronization overhead, memory divergences, and branch divergences. We can observe that most benchmarks exhibit performance degradation when running large warps where NK-threaded warps running on K-wide SIMD lanes. Static large warp architectures cannot adapt to the varying thread-level parallelism among applications. For example, in the default 32-threaded warp architecture, if all threads on an SM take one direction and all threads on another SM take another direction, they can execute concurrently. However, by grouping the same threads into a larger 64-threaded or 128-threaded warp, they will be serialized, causing performance loss. This is especially true for benchmarks optimized for the 32-threaded warp architecture. For two-buddy or four-buddy configurations, it initially incurs the same condition as a 64- or 128-threaded warp architecture but is immediately ungrouped after the first divergence. After that, it works just as the 32-threaded warp architecture. Buddy SM incurs at most one-time divergence in each kernel with the "new kernel" regrouping strategy, but large warp architectures stall for every divergence in each kernel.

Overall, the performance of two-buddy or four-buddy configurations is close to that of the default system. We can observe that the performance of Buddy SM decreases with the buddy size, since bigger buddy sizes may introduce more stalls and synchronization

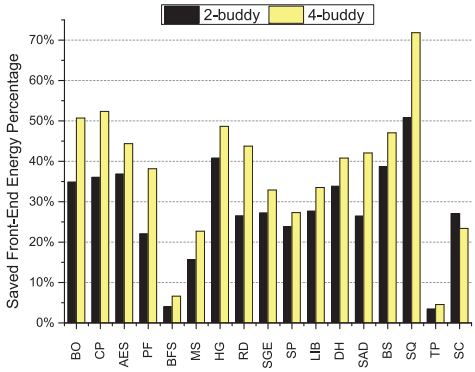

Fig. 20. Front-end energy savings.

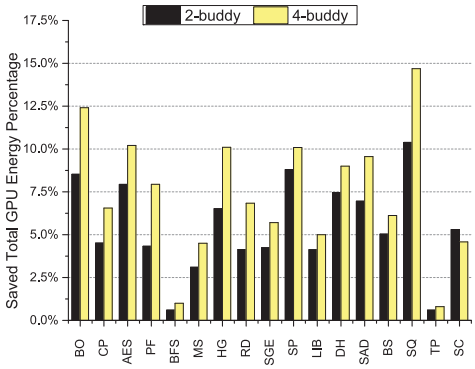

Fig. 21. Total GPU energy savings.

cost because of more SMs in each buddy cluster. Eight-buddy configurations exhibit significant performance degradations, and therefore we recommend buddy sizes of four or smaller.

6.6. Front-End Energy Savings

Figure 20 presents the energy saved at the SM front-end units for the two-buddy and four-buddy configurations, respectively. The results correspond to the energy savings minus the overhead. The energy savings come from the power-gated front-end units of the slaves in the Buddy SM execution. The dynamic power and the static power of the front-end units are acquired from GPUWattch at runtime. The energy overhead includes the energy consumed on the buddy cluster NoC, enhanced scoreboard, and energy due to the prolonged execution time for some applications. Generally, a larger front-end energy savings can be achieved if an application has larger buddy time percentage and better performance. We can see that all applications save energy in both the two-buddy and four-buddy cases. Generally, the four-buddy case saves more energy because more front-end units of slaves can be power gated. It is worthwhile to mention that the four-buddy case has larger energy overhead due to its slightly reduced performance than the two-buddy case. In addition, the energy spent on the cluster NoC is larger for the four-buddy base because the packets need to travel longer distances. On average, 27.8% and 37.2% front-end energy can be saved under two-buddy and four-buddy configurations, respectively. We do not advocate cluster size beyond four because the performance/power is not good. SQ achieves the best front-end energy savings because of its good performance and 100% buddy time. On the contrary, BFS and TP save only a poor portion of the front-end energy because of their small buddy time or bad performance. SC saves more energy in two-buddy configurations than in four-buddy configurations because of its much better performance in two-buddy configurations, as show in Figure 19. During the prolonged execution time in four-buddy cases, the whole GPU consumes power, which offsets the energy savings. As a result, SC saves more energy in two-buddy than in four-buddy configurations.

6.7. Total GPU Energy Savings

Figure 21 presents the results for the total saved energy (the energy savings minus the overhead) of the whole GPU. The energy savings percentage for the entire GPU depends on three factors: application performance (Figure 19), buddy time percentage (Figure 14), and the percentage of the front-end power in total GPU power. We can see that the four-buddy scheme saves more energy than the two-buddy scheme for all

applications except SC. For SC, its worse performance in the four-buddy scheme and the corresponding energy overhead outweigh the extra energy saved from more slave SMs in the four-buddy scheme. Overall, SQ saves the highest energy percentage, whereas BFS and TP save the least energy percentage. Five applications save more than 10% of total energy. From the figure, we can see that compute-intensive applications (BO, CP, AES, PF), memory-intensive applications (BS, SQ, TP, SC), and some irregular applications (BFS, MS, HG) all benefit from Buddy SM for energy efficiency. On average, 5.4% and 7.5% total GPU energy savings are obtained for all applications under two-buddy and four-buddy configurations, respectively. In summary, the two-buddy scheme has slightly better performance, whereas the four-buddy scheme saves more energy. Choosing which buddy size is a trade-off between performance and energy. We suggest using a static buddy size on a GPU.

7. CONCLUSION

In this article, we propose Buddy SM, a front-end sharing scheme to improve the energy efficiency in GPGPUs. Multiple SMs can be grouped together and execute in the lock-step fashion in a buddy cluster. The working principle is based on the unique characteristics in GPGPUs that many thread blocks behave similarly during the program execution, and thus there is no need for duplicated front-end units and independent operations. Buddy SM can save up to 37.2% front-end energy and 7.5% total GPU energy, respectively. The experiments show that Buddy SM is effective in both compute-intensive and memory-intensive applications. In addition, some irregular applications can also benefit from the Buddy SM scheme for energy efficiency.

For future work, we will improve the Buddy SM technique for better efficiency. In addition, we will design new warp scheduling algorithms that can better cope with the Buddy SM execution and produce fewer stalls. We will seek for more efficient buddy strategies and study the feasibility of this technique to other types of processors.

ACKNOWLEDGMENTS

The authors would like to thank anonymous reviewers for their fruitful feedback and comments that have helped to improve the quality of this work.

REFERENCES

Tor M. Aamodt and Wilson W. L. Fung. 2013. GPGPU-Sim 3.x Manual. Retrieved February 1, 2015, from http://gpgpu-sim.org/manual/index.php/GPGPU-Sim_3.x_Manual.

Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'09)*. IEEE, Los Alamitos, CA, 163–174. DOI:http://dx.doi.org/10.1109/ISPASS.2009.4919648

Nathan Brookwood. 2010. AMD Fusion Family of APUs: Enabling a Superior, Immersive PC Experience. Retrieved February 1, 2015, from http://www.amd.com/Documents/48423_fusion_whitepaper_WEB.pdf.

Martin Burtscher, Rupesh Nasre, and Keshav Pingali. 2012. A quantitative study of irregular programs on GPUs. In *Proceedings of the 2012 IEEE International Symposium on Workload Characterization (IISWC'12)*. IEEELos Alamitos, CA, 141–151. DOI:http://dx.doi.org/10.1109/IISWC.2012.6402918

Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'09)*. IEEE, Los Alamitos, CA, 44–54. DOI:http://dx.doi.org/10.1109/IISWC.2009.5306797

Thomas H. Dunigan Jr., Jeffrey S. Vetter, James B. White III, and Patrick H. Worley. 2005. Performance evaluation of the Cray X1 distributed shared-memory architecture. *IEEE Micro* 25, 1, 30–40. DOI:http://dx.doi.org/10.1109/MM.2005.20

Wilson W. L. Fung and Tor M. Aamodt. 2011. Thread block compaction for efficient SIMT control flow. In *Proceedings of the IEEE 17th International Symposium on High-Performance Computer Architecture (HPCA'11)*. IEEE, Los Alamitos, CA, 25–36. DOI:http://dx.doi.org/10.1109/HPCA.2011.5749714

Wilson W. L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. 2007. Dynamic warp formation and scheduling for efficient GPU control flow. In *Proceedings of the IEEE/ACM 45th Annual International Symposium on Microarchitecture (MICRO'07)*. IEEE, Los Alamitos, CA, 407–420. DOI: http://dx.doi.org/10.1109/MICRO.2007.12

Wilson W. L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. 2009. Dynamic warp formation: Effcient MIMD control flow on SIMD graphics hardware. *ACM Transactions on Architecture and Code Optimization* 6, 2, 1–37. DOI: http://dx.doi.org/10.1145/1543753.1543756

Mark Gebhart, Daniel R. Johnson, David Tarjan, Stephen W. Keckler, William J. Dally, Erik Lindholm, and Kevin Skadron. 2011. Energy-efficient mechanisms for managing thread context in throughput processors. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA'11)*. ACM, New York, NY, 235–246. DOI: http://dx.doi.org/10.1145/2000064.2000093

Nilanjan Goswami, Bingyi Cao, and Tao Li. 2013. Power-performance co-optimization of throughput core architecture using resistive memory. In *Proceedings of the 2013 IEEE 19th International Symposium on High-Performance Computer Architecture (HPCA'13)*. IEEE, Los Alamitos, CA, 342–353. DOI: http://dx.doi.org/10.1109/HPCA.2013.6522331

Sunpyo Hong and Hyesoon Kim. 2010. An integrated GPU power and performance model. In *Proceedings of the International Symposium on Computer Architecture (ISCA'10)*. ACM, New York, NY, 280–289. DOI: http://dx.doi.org/10.1145/1815961.1815998

Engin Ipek, Meyrem Kirman, Nevin Kirman, and Jose F. Martinez. 2007. Core fusion: Accommodating software diversity in chip multiprocessors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA'07)*. ACM, New York, NY, 186–197. DOI: http://dx.doi.org/10.1145/1250662.1250686

Naifeng Jing, Yao Shen, Yao Lu, Shrikanth Ganapathy, Zhigang Mao, Minyi Guo, Ramon Canal, and Xiaoyao Liang. 2013. An energy-efficient and scalable eDRAM-based register file architecture for GPGPU. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA'13)*. ACM, New York, NY, 344–355. DOI: http://dx.doi.org/10.1145/2485922.2485952

Changkyu Kim, Simha Sethumadhavan, M. S. Govindan Nitya, Ranganathan, Divya Gulati, Doug Burger, and Stephen W. Keckler. 2007. Composable lightweight processors. In *Proceedings of the IEEE/ACM Annual International Symposium on Microarchitecture (MICRO'07)*. IEEE, Los Alamitos, CA, 381–394. DOI: http://dx.doi.org/10.1109/MICRO.2007.10

Ahmad Lashgar, Amirali Baniasadi, and Ahmad Khonsari. 2012. Dynamic warp resizing: Analysis and benefits in high-performance SIMT. In *Proceedings of the IEEE 30th International Conference on Computer Design (ICCD'12)*. IEEE, Los Alamitos, CA, 502–503. DOI: http://dx.doi.org/10.1109/ICCD.2012.6378694

Ahmad Lashgar, Amirali Baniasadi, and Ahmad Khonsari. 2013. Inter-warp instruction temporal locality in deep-multithreaded GPUs. In *Proceedings of the 26th International Conference on Architecture of Computing Systems (ARCS'03)*. 134–146. DOI: http://dx.doi.org/10.1007/978-3-642-36424-2_12

Jingwen Leng, Syed Gilani, Tayler Hetherington, Ahmed ElTantawy, Nam Sung Kim, Tor M. Aamodt, and Vijay Janapa Reddi. 2013. GPUWattch: Enabling energy optimizations in GPGPUs. In *Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA'13)*. ACM, New York, NY, 487–498. DOI: http://dx.doi.org/10.1145/2485922.2485964

Jiayuan Meng, Jeremy W. Sheaffer, and Kevin Skadron. 2012. Robust SIMD: Dynamically adapted SIMD width and multi-threading depth. In *Proceedings of the IEEE 26th International Parallel and Distributed Processing Symposium (IPDPS'12)*. IEEE, Los Alamitos, CA, 107–118. DOI: http://dx.doi.org/10.1109/IPDPS.2012.20

Jiayuan Meng, David Tarjan, and Kevin Skadron. 2010. Dynamic warp subdivision for integrated branch and memory divergence tolerance. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA'10)*. ACM, New York, NY, 235–246. DOI: http://dx.doi.org/10.1145/1815961.1815992

Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P. Jouppi. 2009. CACTI 6.0: A Tool to Model Large Caches. Retrieved February 1, 2015, from http://www.hpl.hp.com/techreports/2009/HPL-2009-85.pdf

Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N. Patt. 2011. Improving GPU performance via large warps and two-level warp scheduling. In *Proceedings of the IEEE/ACM 45th Annual International Symposium on Microarchitecture (MICRO'11)*. ACM, New York, NY, 308–317. DOI: http://dx.doi.org/10.1145/2155620.2155656

NVIDIA Corporation. 2009. NVIDIA's Next Generation CUDA Compute Architecture: Fermi. Retrieved February 1, 2015, from http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.

NVIDIA Corporation. 2012a. NVIDIA CUDA Toolkit 4.1—Archive. Retrieved February 1, 2015, from https://developer.nvidia.com/cuda-toolkit-41-archive

NVIDIA Corporation. 2012b. NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110. Retrieved February 1, 2015, from http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf.

Minsoo Rhu and Mattan Erez. 2012. CAPRI: Prediction of compaction-adequacy for handling control-divergence in GPGPU architectures. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA'12)*. IEEE, Los Alamitos, CA, 61–71. DOI:http://dx.doi.org/10.1109/ISCA.2012.6237006

Minsoo Rhu and Mattan Erez. 2013a. The dual-path execution model for efficient GPU control flow. In *Proceedings of the IEEE 19th International Symposium on High-Performance Computer Architecture (HPCA'13)*. IEEE, Los Alamitos, CA, 591–602. DOI:http://dx.doi.org/10.1109/HPCA.2013.6522352

Minsoo Rhu and Mattan Erez. 2013b. Maximizing SIMD resource utilization in GPGPUs with SIMD lane permutation. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA'13)*. ACM, New York, NY, 356–367. DOI:http://dx.doi.org/10.1145/2485922.2485953

Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. 2012. Cache-conscious wavefront scheduling. In *Proceedings of the IEEE/ACM 45th Annual International Symposium on Microarchitecture (MICRO'12)*. IEEE, Los Alamitos, CA, 72–83. DOI:http://dx.doi.org/10.1109/MICRO.2012.16

Tarinder Sandhu. 2010. NVIDIA's GeForce GTX 480 Finally Unleashed. Reviewed and Rated. Retrieved February 1, 2015 from http://hexus.net/tech/reviews/graphics/24000-nvidias-geforce-gtx-480-finally-unleashed-reviewed-rated/?page=2.

John A. Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen Mei W. Hwu. 2012. *Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing*. IMPACT Technical Report, IMPACT-12-01, University of Illinois Urbana-Champaign, Champaign, IL.

David Tarjan, Michael Boyer, and Kevin Skadron. 2008. Federation: Repurposing scalar cores for out-of-order instruction issue. In *Proceedings of the 45th ACM/IEEE Design Automation Conference (DAC'08)*. ACM, New York, NY, 772–775. DOI:http://dx.doi.org/10.1145/1391469.1391666

Theo Valich. 2010. NVIDIA "Fermi" GeForce Die Sizes Exposed. Retrieved February 1, 2015, from http://www.brightsideofnews.com/news/2010/8/9/nvidia-fermi-geforce-die-sizes-exposed.aspx.

Wing-Kei S. Yu, Ruirui Huang, Sara Q. Xu, Sung-En Wang, Edwin Kan, and G. Edward Suh. 2011. 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 (ISCA'11)*. ACM, New York, NY, 247–258. DOI:http://dx.doi.org/10.1145/2000064.2000094

Tao Zhang and Xiaoyao Liang. 2014. Dynamic front-end sharing in graphics processing units. In *Proceedings of the 32nd IEEE International Conference on Computer Design (ICCD'14)*. IEEE, Los Alamitos, CA, 286–291. DOI:http://dx.doi.org/10.1109/ICCD.2014.6974695

Ying Zhang, Yue Hu, Bin Li, and Lu Peng. 2011. Performance and power analysis of ATI GPU: A statistical approach. In *Proceedings of the 6th IEEE International Conference on Networking, Architecture, and Storage (NAS'11)*. IEEE, Los Alamitos, CA, 149–158. DOI:http://dx.doi.org/10.1109/NAS.2011.51

Hongtao Zhong, Steven A. Lieberman, and Scott A. Mahlke. 2007. Extending multicore architectures to exploit hybrid parallelism in single-thread applications. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA'07)*. IEEE, Los Alamitos, CA, 25–36. DOI:http://dx.doi.org/10.1109/HPCA.2007.346182

Received October 2014; revised February 2015; accepted March 2015