Temperature Aware Thread Block Scheduling in GPGPUs

Rajib Nath
University of California, San Diego
rknath@ucsd.edu

Raid Ayoub
Strategic CAD Labs, Intel Corporation
raid.ayoub@intel.com

Tajana Simunic Rosing
University of California, San Diego
tajana@ucsd.edu

ABSTRACT

In this paper, we present a first general purpose GPU thermal management design that consists of both hardware architecture and OS scheduler changes. Our techniques schedule thread blocks from multiple computational kernels in spatial, temporal, and spatio-temporal ways depending on the thermal state of the system. We can reduce the computation slowdown by 60% on average relative to the state of the art techniques while meeting the thermal constraints. We also extend our work to multi GPGPU cards and show improvements of 44% on average relative to existing technique.

1. INTRODUCTION

General purpose graphics processor unit ($GPGPU$) provides an energy efficient computing platform for a wide range of parallel applications. The rising trend in the number of cores per $GPGPU$ chip, in addition to technology scaling, results in high power densities. High power dissipation causes thermal hot spots that may have a significant effect on reliability, performance, and leakage power [2]. Meanwhile the duty cycle time of $GPGPUs$ is getting shorter because of the advancements in the $PCIe$ bus design, $GPGPU's$ ability to hide the data transfer with overlapping computation, and $GPGPU$ multiuser mode. As a result, it is becoming more challenging to dissipate the heat using existing cooling mechanisms without sacrificing performance.

In $NVIDIA$ $GPGPUs$, the submitted jobs, usually referred to as kernels, wait in a queue called *KQueue*. Each kernel has a massive number of threads, which are divided into disjoint groups called thread blocks (TB). A kernel has hundreds to thousands of TBs, which have a very short lifetime (μs to ms). Threads inside each TB may synchronize, though TBs are completely independent of each other. TBs are scheduled to the available streaming multiprocessors (SM) by a hardware TB scheduler. Kernels from *KQueue* are processed one at a time unless there are unused SMs to launch more TB from the next kernel. Since all concurrently running TBs in modern $GPGPUs$ are usually from a single kernel, their temperature is dominated by that kernel. A highly compute intensive kernel occupying all the cores can increase the temperature very quickly, causing performance degradation due to dynamic thermal management (DTM) techniques like throttling or dynamic voltage frequency scaling ($DVFS$).

In this work, we propose temperature aware TB scheduling (T^ABS), the first ever $GPGPU$ specific thermal management technique. T^ABS exploits the data parallel computation pattern of $GPGPUs$, the short life time of TBs, the abundance of TBs, and the thermal heterogeneity in $GPGPU$ workloads to reduce the thermal hotspots. T^ABS intelligently intermixes kernels without doing any thread migration to eliminate the overhead of context switching whereas most of the thermal aware workload scheduling in CPU require thread migrations. We present three classes of intermixing algorithms for T^ABS: (a) temporal (alternate), (b) spatial (mixed), (c) spatio-temporal (mixed-alternate). T^ABS has great opportunities in new $GPGPUs$ like Kepler where kernels from multiple applications submitted by more than one user can run concurrently in a virtualized $GPGPU$ environment. We provide the necessary architectural and software changes which make it possible to implement T^ABS in $GPGPUs$. We also explore the use of T^ABS in multi-$GPGPU$ graphics cards. We present a thorough evaluation of T^ABS which reduces performance cost due to thermal hotspots by 60% on average, while meeting thermal constraints.

2. RELATED WORK

Thermal management for general purpose processors has been an active research area in recent years. There are two classes of core level thermal management techniques: reactive and proactive. Popular reactive dynamic thermal management (DTM) [5] techniques remove the excess heat aggressively by slowing down the cores by using pipeline throttling or $DVFS$ at associated performance cost, which our proposed technique reduces dramatically. Activity migration has also been proposed to manage the excess temperature by rescheduling computation across redundant units [6, 9, 8]. This technique does not fit well in the context of $GPGPU$ programming model due to the thread structure, the computation pattern, and the migration cost in short life time of the threads. Moreover, migrations of a kernel across $GPGPUs$ may involve lots of data transfer thereby increasing the performance and energy overhead. In order to address the performance overhead and non uniform ther-

DAC '13, May 29 - June 07 2013, Austin, TX, USA

mal distribution in reactive techniques, a set of proactive thermal management techniques [7, 16] have been introduced that leverage temperature predictors. T^ABS can be complemented farther by such temperature predictors if future $GPGPU$ kernels show heterogeneous thermal phases in a single run. Sheaffer, et al. [13] has explored multiple thermal management techniques such as global clock gating, fetch gating, dynamic voltage scaling (DVS), and DVS in multiple clock domains for graphics workloads in $GPUs$. However, all these techniques have performance slowdown due to the induced throttling. Despite all the existing work in $CPUs$ and recent comparison of DTM techniques for graphics workloads in $GPUs$, the thermal management of $GPGPUs$ has not been explored. Our work exploits the thermal heterogeneity in $GPGPU$ kernels and intermixes thread blocks from multiple kernels to have a better thermal distribution over time and space. Although spatial scheduling has been recently proposed for $GPGPUs$ to maximize the resource usage that ultimately leads to increased performance [1], it does not take temperature into account; as a result it may co-schedule two hot kernels, thus experiencing performance overhead due to thermal hotspot.

3. T^ABS DESIGN

We motivate our work by running $GPGPU$ workloads on $NVIDIA$'s $GTX280$ [12] graphics card which has a maximum total power consumption of $236W$ (more in Section 4.1). We measure temperature difference as high as $27°C$ when running different kernels, with the hottest kernel running at the maximum allowed temperature with the $GPGPU$ fan running at the maximum operating speed. Interleaving the execution of the hottest and coldest kernels lowers the measured temperature by $15°C$ below maximum. To investigate the performance overhead caused by thermal hotspots, we develop a thermal simulator for $GTX280$ (details in Section 4) and run a micro kernel that consumes $120W$ power in the SMs. In the execution period of $800s$, we have found that all the SMs occasionally reach the critical temperature and thus experiences 22% performance overhead as a result of the $GPGPU$'s default throttling mechanism. We observe no throttling when running a low power kernel that consumes a total of $60W$ SM power. Interleaving these two kernels with $100ms$ time interval leads to 6% overhead, a large reduction. Since throttling rate relates to the SM temperature, the kernel alternation pattern eventually lowers the temperature. Time multiplexing is beneficial from the thermal and performance points of view when there is no context switching overhead. While such overhead can be very large in general purpose processor, we show that this is negligible and avoidable in $GPGPUs$. We observe similar thermal benefits by distributing available SMs among hot and cold kernels (SM multiplexing). These results have motivated us to intermix TBs from thermally heterogeneous kernels and to modify the existing HW TB and OS schedulers.

3.1 T^ABS System Architecture

Our proposed system architecture for temperature aware thread block scheduler (T^ABS), shown in Figure 1, has two components: HW TB scheduler and *OS scheduler*. They coordinate with each other to execute temperature aware TB scheduling algorithms in order to reduce thermal hotspots in $GPGPUs$. HW TB scheduler maintains power and lifetime

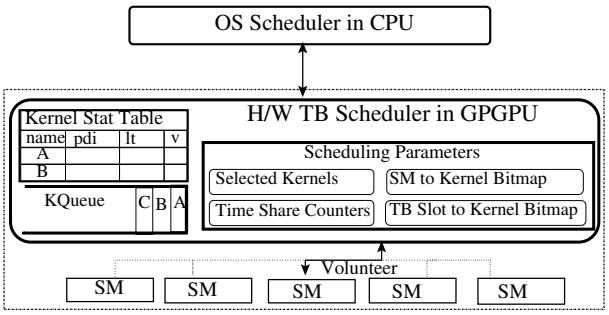

Figure 1: T^ABS Architecture

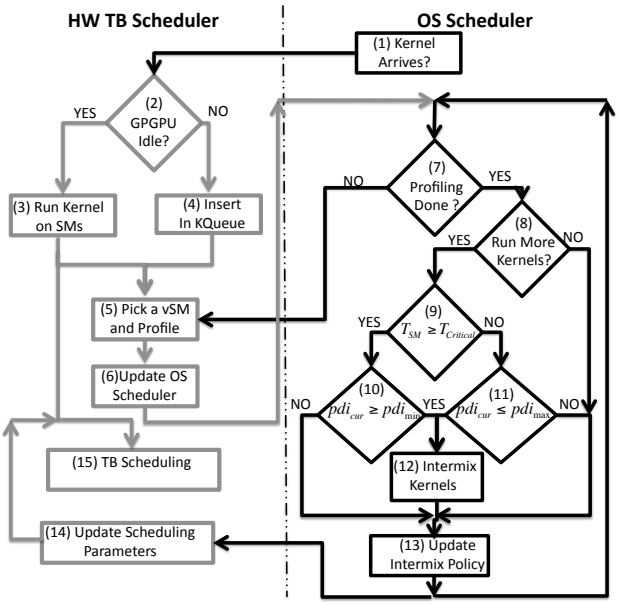

Figure 2: Interactions in T^ABS

statistics of TBs of each kernel in a kernel stat table. The key to determining which kernel should run next is comparing the currently running kernel with kernels in $KQueue$ in terms of power density, and therefore the chance to reduce the temperature. *OS scheduler* uses the kernel stat table along with the instantaneous temperatures of SMs to set the TB scheduling policy for the next OS scheduling tick. *OS scheduler* periodically updates HW TB scheduler about the set of intermixed kernels and their relative share of $GPGPU$ resources (e.g., time, SMs), which define and parametrize various intermixing policies for T^ABS.

Figure 2 shows the interaction between HW TB scheduler and *OS scheduler*. When a $GPGPU$ job arrives in our system (step 1), the *OS scheduler* sends it to $GPGPU$. In step 2 when the $GPGPU$ is idle, the arriving kernel starts execution immediately. Otherwise the kernel waits in the $KQueue$ (step 4), which is usually served in the first come first served ($FCFS$) manner. As soon as a kernel arrives in $KQueue$, the TB scheduler selects a volunteer SM (vSM) to profile the new kernel (step 5). The coldest SM in the $GPGPU$ is selected as vSM. There can be multiple $vSMs$ simultaneously profiling multiple kernels if several kernels arrive in $KQueue$ in a short duration of time. The number of $vSMs$ cannot be more than the number of SMs in the $GPGPU$. The TB scheduler waits for the running TBs in a vSM to finish before scheduling a batch of TBs from that kernel.

The profile of each kernel, provided by vSM, is stored in a kernel stat table. Each entry in the kernel stat table has four fields: (a) name, (b) power density index, (c) average TB lifetime, (d) valid bit. Power density index (*pdi*) is an estimate of the average dynamic power consumption of one SM while running TBs from that kernel. We use the model in Equation 1, derived from [10], to estimate the *pdi* based on activity rates in integer unit (INT), floating point unit (FP), special function unit (SFU), cache ($CACHE$), register file (REG), shared memory (SHM), arithmetic logic unit (ALU), and fetch decode unit (FDS)

$$P_{SM} = \sum_{i \in Units} P_i = P_{INT} + P_{FP} + P_{SFU} + P_{ALU} + P_{FDS} + P_{REG} + P_{CACHE} + P_{SHM} \quad (1)$$

where $P_i = MaxPower_i \times AccessRate_i$ for each unit i. This model has the sufficient accuracy to compare thermal behavior of two kernels based on power consumption [10]. The performance monitoring unit (PMU) in the vSM monitors the performance counters in Equation 1 until one of the TB from the batch retires. This way the monitoring is performed with a constant thread level parallelism (TLP) in the vSM which is necessary for an accurate estimate of the *pdi* in $GPGPUs$. Once the activity data and average lifetime (lt) of TBs are available from the vSM, the TB scheduler computes the *pdi*, updates the corresponding kernel entry in the kernel stat table and marks it as profiled by setting the valid bit(v). One time sampling of *pdi* is adequate to classify the power profile of the kernels because the dynamic power consumption of $GPGPU$ applications is mostly constant as shown in many publications [10]. *OS scheduler* uses *pdi* to assess whether a particular kernel will run hotter or colder on average than a competitor kernel. Since *pdi* is unbiased with respect to the thermal history, the current temperature of the SMs, and the activity in the neighboring SMs, it is a more reliable metric than direct temperature sensor reading.

Once the TB scheduler informs the *OS scheduler* that the kernels in the $KQueue$ are profiled (steps 6 & 7), the *OS scheduler* looks for intermixing opportunities in every OS scheduling tick ($50ms$) (steps 8 through 12). When the $GPGPU$ has space for running more parallel kernels (step 8), the *OS scheduler* explores $KQueue$ for the best option while checking the thermal state of the $GPGPU$ (step 9). During a thermal emergency situation when the SM temperature reaches emergency threshold (step 10), the *OS scheduler* intermixes the pending kernels with the lowest *pdi* with the running kernel if possible (step 10) and assigns an initial small percentage (e.g., 5%) of total $GPGPU$ resources (e.g., time, SMs) to it (step 13). The *OS scheduler* takes proactive action in the absence of temperature problem by intermixing the maximum *pdi* pending kernel with the running kernel whenever possible (step 11). Such decision is beneficial because it is ideal to spread the heat uniformly over time. When more than one kernel is running in a thermal emergency, the *OS scheduler* updates the policy with a new distribution of resources if needed (step 13).

Each time the SM temperature reaches a critical value, the *OS scheduler* progressively takes away resources from the kernel with higher *pdi* and assigns those to the kernel with a lower *pdi*. Although taking away resources from kernels may affect the order at which kernels finish, this helps to increase the overall $GPGPU$ throughput in the presence of thermal emergencies. As the thermal problem disappears, the *OS scheduler* does the reverse to ensure that the victim kernel with a high *pdi* is not falling behind in execution. The minimum resource assigned to the victim kernel that arrived first in the $KQueue$ among two concurrently running kernels does not go below 50% of the total $GPGPU$ resources. This way, the *OS scheduler* prevent starvation of the kernels with high *pdi*. In order to ensure fairness, the *OS scheduler* maintains a non-overlapping window of N kernels that can be intermixed. The window size represents a trade-off between fairness and intermixing opportunity. For example, T^ABS will act as a first come first serve ($FCFS$) scheduler that performs no intermixing when $N = 1$. While we pick $N = 32$ to match $KQueue$ size, any smart fairness algorithm can be implemented with this framework.

The *OS scheduler* updates the $HW\ TB$ scheduler with new scheduling parameters (step 14). The scheduling parameters, describing the distribution of resources (time and SM) among intermixed kernels, defines different classes of intermixing policies: (a) alternate, (b) mixed, and (c) mixed alternate. Alternate (A) intermixing policy time shares the GPGPU between multiple heterogeneous kernels in an interleaved fashion. The TB scheduler executes the alternation of kernels inside each $GPGPU$ while the time sharing interval of each kernel is dynamically determined by the *OS scheduler* at each OS scheduling tick. The time sharing interval of each kernel should be an integer multiple of the lifetime of its TB. In this way, the T^ABS does not interrupt execution to maximize performance while keeping the same time share ratio. With mixed intermixing policy, the TB scheduler schedules multiple heterogeneous kernels with different power densities on a single $GPGPU$ simultaneously. Mixed intermixing is further divided into two subtypes depending on the heterogeneity of TBs in a single SM. In mixed uniform (MU), each SM is allowed to host TBs from a single kernel. The number and the topological location of SMs that run TBs from each kernel is an optimization parameter and is dynamically determined by the *OS scheduler* depending on the *pdi* of individual kernels and the thermal state of the system. T^ABS usually schedules TBs with higher *pdi* in the corner SMs. The *OS scheduler* updates the $HW\ TB$ scheduler with a SM to kernel bitmap which describes the SM distribution. Mixed nonuniform (MNU) allows TBs from multiple kernels to coincide together in a single SM. The *OS scheduler* distributes available TB slots in each SM among the concurrently running heterogeneous kernels and updates the TB slot to kernel bitmap. Since SMs will have heterogeneous threads, MNU is analogous to alternate at finer granularity. Mixed alternate (MA) is a hybrid technique that uses mixed and alternate approaches at the same time. Initially MA allocates x SMs to a *hot* kernel and $(M - x)$ SMs to a cold kernel where M is the total number of SMs in a GPGPU. In the subsequent alternating intervals, each SM alternates between hot and cold kernels.

For each intermixing policy, TB scheduler allows the running TBs to finish before adopting to new scheduling parameters provided by the *OS scheduler* (step 15). Since TBs have very short lifetime (μs to few ms) and the thermal time constraint of $GPGPU$ is usually in the range of seconds [14], delaying the update is not an issue. The short lifetime of TBs, the abundance of TBs, and TB scheduler's ability to schedule new TBs with near zero context switch overhead have allowed us to use T^ABS as an efficient thermal

management technique for modern $GPGPUs$. This unique property of $GPGPU$ has given us the freedom to manage the thermal problems of $GPGPU$ without doing any context switching or migration of threads which are normally the key techniques of temperature management in $CPUs$. The same techniques cannot be applied to $CPUs$ because CPU threads have a much longer lifetime that often exceeds the thermal time constant. A scheduler in the cluster level or supercomputer level can reuse the power profiles of each kernel to maintain a heterogeneous mixture of kernels in the intermixing window of each compute node. In the absence of heterogeneous kernels in the KQueue, T^ABS applies $DVFS$ to handle thermal emergency situations if required.

3.2 T^A BS Across Multiple GPGPUs

In a multi-$GPGPU$ graphics card (e.g., $NVIDIA\ GTX$ 690), a single fan is shared by two $GPGPUs$, so the fan speed rises with the temperature of the hottest $GPGPU$ chip. Instead of executing two heterogeneous kernels (*hot* and *cold*) in two separate $GPGPUs$, TBs from each kernel can be distributed intelligently among them. This also saves cooling energy since the cooling cost is cubically related to the fan speed. *OS schedulers* contacts TB scheduler in all the $GPGPUs$ to find a heterogeneous mix of kernels. Similar to a single $GPGPU$ case, the percentage of TBs from concurrently running kernels in each $GPGPU$ is an optimization parameter that the *OS scheduler* sets dynamically.

3.3 T^A BS Overhead

In this section, we discuss both area and performance overhead of our techniques. Area overhead is dominated by the kernel stat table whose size is bound by the size of $KQueue$ (e.g., 32). The overhead of maintaining kernel statistics is 832 bits where each entry needs 26 bits: 5 bits for kernel name, 4 bits for *pdi*, 16 bits for *lt* and 1 bit for *v*. To record the resource sharing info for different intermixing policies, TB scheduler also needs two counters to keep track of the time quantum progress of intermixed kernels, one bit map (bounded by number of SM in the chip) to map SM to kernels, and one common bit map for all SMs (bounded by 8) to map TB slot per SM to kernels. These additional hardware requirements (2 counters + 30 bits + 8 bits) also represent a very low overhead.

Since the lifetime of each TB is typically in the range of μs to few ms and each kernel is profiled once, the kernel profiling overhead is very small. TB scheduler launches a system kernel (one TB with 8 threads) which computes *pdi* from the PMU activity data using using Equation 1. TB scheduler also uses two other system kernels (one TB with 32 threads) to find the kernels with minimum and maximum *pdi* from the kernel stat table. *OS scheduler* reads these values instead of reading the whole kernel stat table. We use system kernel to save additional HW requirement. Even though the run time of these system kernels is $500ns$ while running alone, the overhead is negligible if executed with other TBs because these are $GPGPU$ reduction (e.g., sum, max, min) kernels with less than 100 instructions. Moreover, they run once in each OS scheduling tick.

T^ABS in multi-$GPGPU$ graphics card has to pay the penalty of accessing memory from the other $GPGPU$ chip on the board. Such overhead is amortized when two kernels with very diverse *pdi* are intermixed.

4. RESULTS

4.1 Methodology

While our solution is applicable to $GPGPUs$ from different vendors, we have selected $NVIDIA$'s $GTX280$ for this study since it is a representative of modern $GPGPUs$ and its power data, power model and floor plan are available from published work [10]. $GTX280$ has 30 streaming multiprocessors (SM), which share a $L2$ cache and an off chip memory. Three SMs share a common L1 cache. Each SM is equipped with 8 $CUDA$ cores, big register bank, and shared local memory. Device memory is interleaved into 8 memory modules providing bandwidth of $140GB/s$. The theoretical peak performance of $GTX280$ is $933GFlops/s$ in single precision at $1.3GHz$ clock speed. The maximum total power of this graphics card is $236W$ while the idle power is $80W$ [10].

Since our contribution requires both architecture and hardware scheduler changes, it is not possible to implement this in today's $GPGPU$ cards. We extend Hotspot [14] simulator with $GPGPU$ thermal simulation based on the floor plan and package characteristics of $GTX280$. The heatsink dimension and the case to ambient thermal resistance (K/W) are used as $0.06m$ and 0.25 respectively based on the data in [15]. Each simulation starts from an initial temperature of $45^\circ C$ with a warm-up period of $200s$. We keep the $GPGPU$ fan running at a fixed speed throughout the simulation to maintain a constant case temperature. There is one temperature sensor per SM. The local ambient temperature within PC and the critical temperature threshold on chip are set at $45^\circ C$ and $90^\circ C$ respectively. Since we are interested in understanding the effectiveness of TB scheduling for thermal management, we focus on SMs only and do not do thermal management of memory or handle cooling. We leverage the power model from [10] to generate the dynamic power traces for our benchmarks. We also account for thermally dependent leakage power based on the model from [3]. Like in [4], we include a baseline power of the $GPGPU$ chip in the simulator.

Bench	SM Power(W)	Bench	SM Power(W)
SVM	87	Bino	67
Sepia	44	Conv	74
Bs	50	Cmem	48
Dotp	60	Madd	61
Dmadd	69	Mmul	64

Table 1: Benchmarks on a GTX 280

We use ten benchmarks [10]. A subset (*SVM*, *Bino*, *Sepia*, *Conv*, *Bs*) is from merge benchmark suite [11] which represents financial and image processing sectors. Others are memory bound (*Dotp*, *Madd*, *Dmadd*, *Mmul*) and compute bound (*Cmem*) benchmarks, which we find in many scientific computing and graphics applications. The dynamic power consumption of SMs while running each benchmark is reported in Table 1. The dynamic range of the SM power consumption is $43W$. We form 10 workloads using these 10 benchmarks to keep a representative mix of hot and cold kernels shown in Table 2. Each benchmark in the workload has computation load of $400s$.

In our simulations, the baseline (BL) DTM policy throttles the $GPGPU$ when the SM temperature reaches the critical threshold and clock gates the $GPGPU$ until the temperature falls below $85^\circ C$. We have also implemented $DVFS$ at five different settings: ($1.18V$, $1.33GHz$), ($1.14V$,

$1.28GHz$), ($1.13V$, $1.26GHz$) , ($1.12V$, $1.20GHz$), and ($1.12V$, $1.15GHz$). We compare T^ABS, the first ever thermal management techniques for $GPGPUs$, against throttling (BL) and $DVFS$, two well known DTM techniques. We show the performance of T^ABS for four different intermixing policies discussed in Section 3.1. T^ABS adopts $DVFS$ if the *OS scheduler* fails to find heterogeneous kernels during a thermal emergency.

Workload	Mix	Workload	Mix
WL1	SVM + Cmem	WL6	Conv+ Bs
WL2	SVM + Bs	WL7	SVM + SVM
WL3	SVM + Dotp	WL8	SVM + Conv
WL4	Conv+ Dmadd	WL9	Conv + Conv
WL5	Conv+ Mmul	WL10	Sepia + Bs

Table 2: Workload Description

4.2 Measured Overhead Due to Intermixing

Running two concurrent kernels using different intermixing policies may cause performance overhead due to sharing of resources, e.g., bandwidth, $L1$ cache, $L2$ cache, and SMs. Due to the alternation of context, the contents of the caches might get flushed. While the impact of cache thrashing can be severe in general purpose $CPUs$, this effect is small in $GPGPUs$ because of the underlying computation patterns. $GPGPU$ kernels perform data parallel computation where multiple threads execute identical code on separate data sets. Instead of depending on high cache hit rates and efficient branch prediction, most of the highly optimized $GPGPU$ kernels depend on the large bandwidth and the high TLP to hide the memory latency.

Intermixing	Performance Improvements (%)		
	Mean	Stdev	Minimum
Alternate	1.26	3.45	-1
Mixed Uniform	5.43	7.35	-1
Mixed Non Uniform	2.83	7.76	-1
Mixed Alternate	2.87	7.01	-1

Table 3: Performance Improvements by Intermixing

Even though new $GPGPUs$ have limited support for parallel kernel execution, they do not intermix TBs. The only way to evaluate the performance overhead due to intermixing TBs from multiple kernels is by manually editing the source code to merge them. In a merged kernel, we keep the functionality of the original kernels unchanged. Inside each merged kernel, threads execute different function based on the $TB\ ID$ and host $SM\ IDs$. For example, in case of mixed uniform intermixing, TB executes function of kernel 1 when $SM\ ID$ is less than 8 and vice versa. The alternate intermixing is simulated by switching between different kernel functions every 100 TBs. Since TB in $GPGPUs$ are scheduled in monotonically increasing order of their $TB\ IDs$, we could emulate our intermixing policies by implementing these tricks. We form 23 different intermixed kernel combinations using benchmarks from Table 1 and run them on actual $GPGPU$. Table 3 shows the mean and standard deviation of performance improvements of all the 23 merged kernels for different intermixing policies. The results show that on average performance improved by 3%, and the worst case overhead is only 1%. Clearly intermixing is beneficial to both performance and on-chip temperature in a large fraction of the cases.

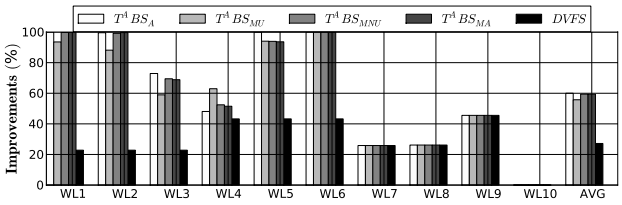

Figure 3: Improvements over BL in Single GPGPU

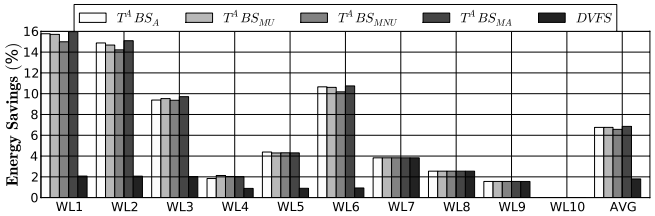

Figure 4: Energy Savings in Single GPGPU

4.3 Simulation Results

Single GPGPU: Our baseline policy (BL), which clock gates the $GPGPU$ until the temperature falls below $85°C$, experiences computation slowdown ranging between 2.06% and 17.79% with an average of 6.75% for the workloads in Table 2. As expected, the performance cost due to throttling is higher for workload with *hot* benchmarks, e.g., $WL7$. Meanwhile, workloads with only *cold* benchmarks (e.g., $WL10$) do not experience any thermal hotspot. Figure 3 shows improvements of T^ABS and $DVFS$ over BL. The improvement is the reduction in computation slowdown due to thermal throttling over the baseline policy. $DVFS$ works better than BL by 27% on average. Our proposed technique T^ABS periodically checks the $KQueue$ and intermix TBs from heterogeneous kernel proactively that results in reduction in number of thermal hotspots and improvement in performance. For different TB intermixing policies: alternate (A), mixed uniform (MU), mixed non uniform (MNU), and mixed alternate (MA) described in Section 3.1, the average reduction in computation slowdown by T^ABS compared to BL ($DVFS$) is from 57%(40%) to 60%(45%). However, for the subset of workloads representing our target cases, T^ABS improves over BL ($DVFS$) by 82%(74%) to 89%(86%). Among the four proposed intermixing policies, alternate works best since it does a good job spreading heat over time. Mixed non-uniform and mixed alternate work better than mixed uniform since mixed uniform runs hot TBs on the same subset of SM while the other two do some kind of alternation of hot and cold kernels.

Figure 4 shows the percentage of energy savings of T^ABS and $DVFS$ relative to BL. Interestingly, all of the intermixing policies of T^ABS save 6.75% on average compared to 1.8% with $DVFS$. The savings of energy come from two different sources. The reduction in computation throttling helps the jobs finish faster. In addition, T^ABS saves leakage power by reducing the average temperature of the SMs. For the heterogeneous workloads representing our target cases ($WL1$, $WL2$, $WL3$, $WL4$, $WL5$, and $WL6$), T^ABS saves 9.48% and 8.12% energy on average relative to BL and $DVFS$ respectively. Energy savings are maximized when we mix the hottest and coldest kernels. For example, the SM power consumption of *Cmem*, *Bs*, *Dotp*, and *Conv* are $48W$, $50W$, $60W$ and $74W$ respectively. When we

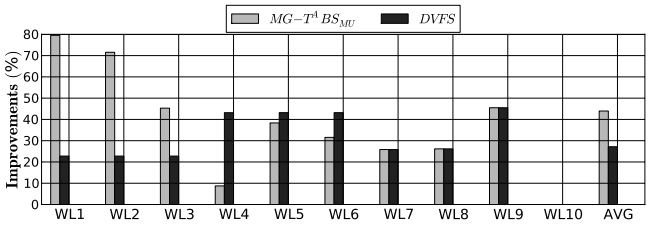

Figure 5: Improvements over BL in Multi GPGPUs

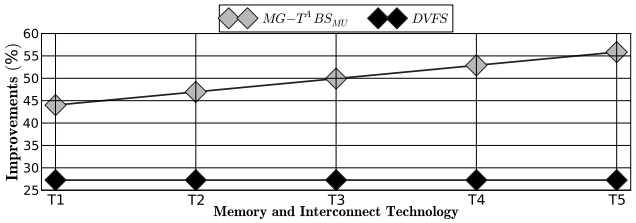

Figure 6: Effects of Memory and Interconnect Technology in Multi $GPGPU$ TB Scheduling Policy

intermix them with SVM, the energy savings are 15.75%, 14.88%, 9.39%, and 1.55% for workload $WL1$, $WL2$, $WL3$, and $WL8$ respectively. In the absence of heterogeneous kernels, the energy savings of our technique is the same as the $DVFS$ ($WL7$ and $WL9$).

Multi GPGPUs: We extend our simulator for graphics card with multiple $GPGPUs$. We use the benchmark data from [10] to estimate memory overhead in our simulator. Figure 5 shows the improvements of $DVFS$, and multi-$GPGPU$ T^ABS ($MG-T^ABS$) with MU intermixing policy over default technique BL. BL and $DVFS$ schedule each kernel to a separate $GPGPU$ and experiences the same amount of throttling that we have seen in single $GPGPU$ case. T^ABS improves over BL by 44% for mixed uniform intermixing policy. Even though the performance of T^ABS is worse than their respective single $GPGPU$ case due to memory overhead, the benefit is still substantial in our target cases, e.g., for $WL1$ T^ABSA improves over BL and $DVFS$ by 80% and 75% respectively.

Results in Figure 5 suggests that scheduling TBs across $GPGPUs$ is beneficial only when the performance gained through T^ABS is greater than the memory overhead. When the thermal profiles of two kernels are very diverse, we get the maximum gain, e.g., $WL1$. For workloads $WL4$, $WL5$ and $WL6$, T^ABS performs worse than $DVFS$ because the benefit gained through reduction in thermal hotspots could not amortize the memory overhead. This observation also implies that as the memory & interconnect technologies in multi-$GPGPU$ cards improve, the gain through T^ABS grows. Figure 6 shows the reduction in computation slowdown for T^ABS and $DVFS$ comparing to BL for different memory & interconnect technologies (from $T1$ to $T5$ memory & interconnect technology gets better). The result suggests that while T^ABS will benefit with memory and interconnect improvements, $DVFS$ will not perform any better than today.

5. ACKNOWLEDGEMENT

This work has been funded by NSF SHF grant 0916127, NSF CCF 1218666, NSF grant 1029783, NSF ERC CIAN EEC-0812072 NSF Variability, CNS, Oracle, Google, Microsoft, MuSyC, and SRC grant P11816.

6. CONCLUSION

The nature of data parallel computation in $GPGPUs$ provides us a unique opportunity to manage the thermal problems by intermixing thread blocks from multiple thermally heterogeneous kernels. In this work, we have proposed T^ABS, the first ever thermal management technique for modern $GPGPUs$, which exploit the opportunity to spread the heat in time and/or space. We have provided the required architectural and software changes to incorporate T^ABS in modern $GPGPUs$. Our results show that T^ABS performs 60% better than state of the art thermal management techniques with energy savings of 6.75% on average. T^ABS's prospect in multi-$GPGPU$ graphics card is also very promising, 44% improvements over well known techniques.

7. REFERENCES

[1] J. T. Adriaens, K. Compton, N. S. Kim, and M. J. Schulte. The case for gpgpu spatial multitasking. *HPCA*, 2012.

[2] A. Ajami, K. Banerjee, and M. Pedram. Modeling and analysis of nonuniform substrate temperature effects on global interconnects. *IEEE Trans. on CAD*, 2005.

[3] R. Ayoub, K. Indukuri, and T. Rosing. Temperature aware dynamic workload scheduling in multisocket cpu servers. *TCAD*, 2011.

[4] R. Ayoub, R. Nath, and T. Rosing. Jetc: Joint energy thermal and cooling management for memory and cpu subsystems in servers. *HPCA*, 2012.

[5] D. Brooks and M. Martonosi. Dynamic thermal management for high-performance microprocessors. *HPCA*, 2001.

[6] J. Choi, C. Cher, H. Franke, H. Hamann, A. Weger, and P. Bose. Thermal-aware task scheduling at the system software level. *ISLPED*, 2007.

[7] A. Coskun, T. Rosing, and K. Gross. Proactive temperature management in mpsocs. *ISLPED*, 2008.

[8] J. Donald and M. Martonosi. Techniques for multicore thermal management: Classification and new exploration. *ISCA*, 2006.

[9] S. Heo, K. Barr, and K. Asanovic. Reducing power density through activity migration. *ISLPED*, 2003.

[10] S. Hong and H. Kim. An integrated gpu power and performance model. *ISCA*, 2010.

[11] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng. Merge: a programming model for heterogeneous multi-core systems. *ASPLOS*, 2008.

[12] NVIDIA. Gtx280 http://www.geforce.com.

[13] J. W. Sheaffer, K. Skadron, and D. P. Luebke. Studying thermal management for graphics-processor architectures. *ISPASS*, 2005.

[14] K. Skadron, M. Stan, K. Sankaranarayanan, W. Huang, S. Velusamy, and D. Tarjan. Temperature-aware microarchitecture: Modeling and implementation. *TACO*, 2004.

[15] J. Wang and W. Chen. Vapor chamber in high-end vga card. *IMPACT*, 2010.

[16] I. Yeo, C. Liu, and E. Kim. Predictive dynamic thermal management for multicore systems. *DAC*, 2008.