An Energy Efficient GPGPU Memory Hierarchy with Tiny Incoherent Caches

Alamelu Sankaranarayanan, Ehsan K. Ardestani, Jose Luis Briz*, and Jose Renau
Dept. of Computer Engineering, University of California Santa Cruz
{alamelu, eka, renau}@soe.ucsc.edu
*Dept. of Computer and System Engineering and I3A, University of Zaragoza
briz@unizar.es

ABSTRACT

With progressive generations and the ever-increasing promise of computing power, GPGPUs have been quickly growing in size, and at the same time, energy consumption has become a major bottleneck for them. The first level data cache and the scratchpad memory are critical to the performance of a GPGPU, but they are extremely energy inefficient due to the large number of cores they need to serve. This problem could be mitigated by introducing a cache higher up in hierarchy that services fewer cores, but this introduces cache coherency issues that may become very significant, especially for a GPGPU with hundreds of thousands of in-flight threads.

In this paper, we propose adding incoherent tinyCaches between each lane in an SM, and the first level data cache that is currently shared by all the lanes in an SM. In a normal multiprocessor, this would require hardware cache coherence between all the SM lanes capable of handling hundreds of thousands of threads. Our incoherent tinyCache architecture exploits certain unique features of the CUDA/OpenCL programming model to avoid complex coherence schemes. This tinyCache is able to filter out 62% of memory requests that would otherwise need to be serviced by the DL1G, and almost 81% of scratchpad memory requests, allowing us to achieve a 37% energy reduction in the on-chip memory hierarchy. We evaluate the tinyCache for different memory patterns and show that it is beneficial in most cases.

Keywords

GPGPUs, Energy-efficiency, Memory hierarchy, Caches

1. INTRODUCTION

General purpose computing on graphics processors is becoming ubiquitous with the acceptance of (and advances in) programming languages like CUDA and OpenCL. This popularity has resulted in the development of GPGPUs like NVIDIA's GeForce GTX 7XX [1] and AMD's Radeon HD 7000 series [2], which are larger than their predecessors and deliver higher performance in the teraflop range. The big challenge however, in progressing toward the next generation of processors, is sustaining performance within a more conservative power budget.

Traditional CPUs are highly latency optimized: They have a few, very complex, high performance cores sharing a pool of memory and are extremely efficient at handling few threads. In contrast, throughput processors like GPGPUs rely on an alternative programming model employing hundreds of very simple cores to execute hundreds of thousands of lightweight threads, to hide long memory latencies. It is this massive multithreading that helps GPGPUs achieve high performance. Several complex (and energy hungry) components are required to make this possible on the GPGPU, including highly banked scratchpad memory and data caches. These caches are shared across multiple processing elements (lanes) within a streaming multiprocessor (SM) on the GPGPU.

This paper focuses on the first level data cache (DL1G) and the scratchpad memory that is shared by all the lanes in an SM. The DL1G is a massive structure, and considering the support it needs to sustain requests from all the lanes and to coalesce memory, it is extremely energy inefficient. For the applications we evaluate, we see that in a typical GPGPU the DL1G accounts for almost 69% of the dynamic energy consumed by the on-chip memory hierarchy. With another 25% expended on the scratchpad memory, this amounts to a total of over 90%, as seen below in Figure 1.

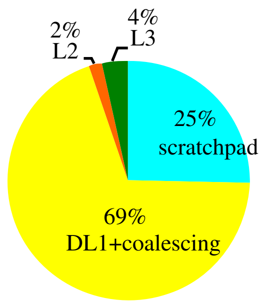

Figure 1: A breakdown of the energy consumption of the on-chip memory hierarchy in a typical GPGPU, for the benchmarks listed in Table 2. The DL1G and the scratchpad memory account for most of the energy consumed.

A seemingly straightforward solution is to add a small filter cache, to intercept frequent requests to the shared DL1G, as described in [3]. The problem is that if we have such a filter cache for each lane in the GPGPU, we will need to maintain coherence between all the lanes in an SM. A CUDA/OpenCL kernel usually spawns thousands of threads that cannot modify the same address without an explicit barrier, but that typically access the same cache line. This is a common strategy used by GPGPU programmers to maximize memory coalescing, which directly impacts the performance (by reducing the memory bandwidth) of the application. This results in a sharing pattern with high false sharing, very different from one seen in typical multiprocessor applications, and makes it highly inefficient to have to maintain coherence between private filter caches per lane.

Instead of simple filter caches, we propose tinyCache, an incoherent, write-back cache *per lane* that leverages an adapted write-validate policy to maintain correctness. A tinyCache cacheline can be merged back to the existing shared L1 cache (DL1G) when the data is displaced, without having to care about invalidations or updates, made possible by the peculiarities of the CUDA/OpenCL programming model. Section 3 provides more details on how we maintain correctness, and why this incoherent behavior can only be supported by the CUDA/OpenCL programming model.

The proposed tinyCache per SM lane filters out a sizable portion of memory accesses to the DL1G. We find that this simple optimization is able to filter out 62% of the memory requests, normally directed toward the DL1G, and about 81% of the requests directed to the scratchpad memory on an average. This results in an average reduction of 37% and 35% in the total energy consumed by the on-chip memory hierarchy and the energy delay product, respectively.

2. BACKGROUND

A typical GPGPU is an array of streaming multiprocessors (SMs), each of which contains a number of simple processing elements (lanes). When a kernel is spawned for execution, groups of threads known as blocks are assigned to individual SMs for execution, and in turn, smaller sets of threads known as warps are scheduled by the warp scheduler to execute on lanes. A warp executes the same instruction across all the lanes on the SM, and continues a lock-step execution until it has to stall due to data dependencies, long latency operations or branch divergence penalties. When it stalls, the warp scheduler schedules another ready warp for execution and thus effectively hides the penalties due to the stalls. The GPGPU typically has plenty of warps ready to take over and it is this massive multithreading that helps the GPGPU tolerate high latency memories.

Threads within a single block communicate with each other using fast on-chip scratchpad memory, but to coordinate with threads of different blocks, they use global memory which is a combination of both on-chip and off-chip memory. In addition to these two memories, threads can also access read-only cached constant and texture memories. There is another type of memory, called local memory, which is private per lane, but is situated in the global space. This local memory is used only in case of register spilling. All the SMs share a common L2 cache which extends to the memory hierarchy below (for *e.g.*, an L3 shared with the CPU in case of an unified CPU-GPU system or to main memory in a system with a discrete GPGPU).

To enable high scalability with respect to the number of cores, the programming model does not allow the programmer to assume a specific order in which thread blocks may be executed. They can be executed in any order, in parallel or in series. To synchronize the memory accesses within a block, there are light weight barriers available, which all threads in a given block must reach before any of them are allowed to proceed further.

Figure 2a shows the internals of a single SM on a modern GPGPU like NVIDIA's Fermi [4] with 32 lanes per SM.

3. MICROARCHITECTURAL CHANGES FOR ENERGY EFFICIENCY

Our main goal is to cut down the large chunk of energy that is spent on costly DL1G accesses. To do this, we propose the addition of tiny incoherent caches per lane in the GPGPU, which serves the single purpose of filtering frequently accessed addresses and avoiding a lookup from the lower levels. As shown in Figure 2b, the tinyCache becomes the first cache in the memory hierarchy for an individual lane. All the global and scratchpad memory requests can be routed through it. To keep the access latency to this cache as small as possible, to maximize its ability to store frequently re-used data and exploit locality, we compare various configurations with a number of entries and different line sizes for the minimum energy delay product. As detailed in Section 5.2, we finally pick a tinyCache with 16 entries and 64B line size.

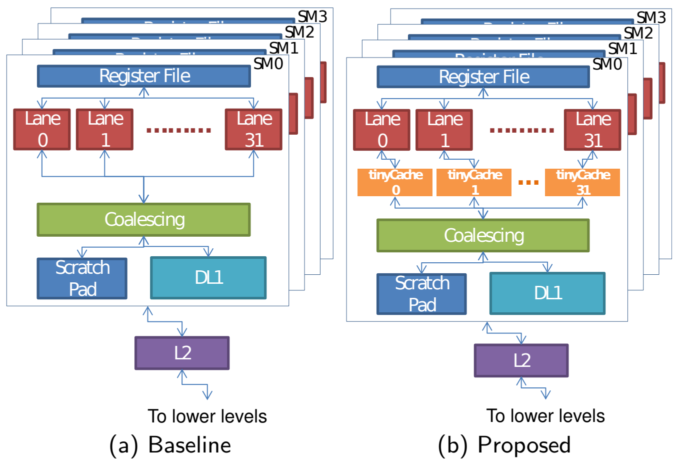

Figure 2: (a) Internals of a single SM in a typical GPGPU. (b) We propose adding tinyCaches per lane. These tinyCaches filter both scratchpad memory and global addresses.

We adopt a write-validate write on-miss policy for the tinyCache [5]. Figure 3 shows the state diagram for a line in the tinyCache,

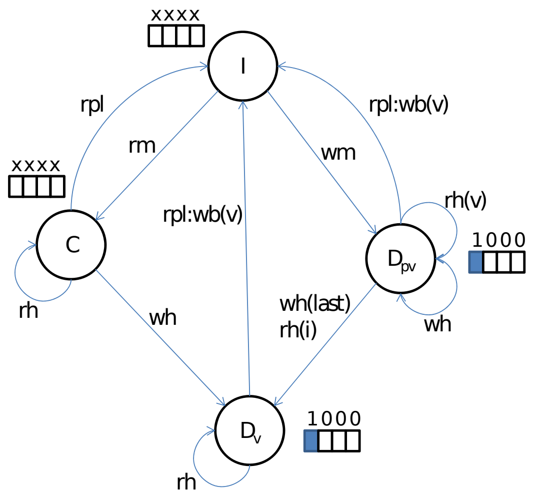

Figure 3: Cache line states and transitions as seen by the tinyCache controller, which implements a write-validate write-on miss policy with write-back update. There is a cache line next to each state as an example. The bits above are the control bits present per half-word (plus two state bits that are not shown). A shaded half-word means that the word was written on entering that state. Note that in D_v, all the half-words are valid (both set and unset control bits denote *valid*), whereas in D_{pv} only the half-words marked *dirty* are valid (unset control bits indicate that the corresponding half-word is are *invalid*).

detailing our implementation of the write-validate policy with write-back. We modify the role of the validity bits in the original protocol, and depending on the state, they can act now as *don't care* bits, validity bits or dirty bits. We will henceforth refer to them as control bits. Our adaptation of the protocol requires four states, which adds two bits per line, plus the control bits, to encode the line state (we omit the transient states here). A control bit per byte would guarantee correct operation with an overhead of 64 bits for a typical 64 Byte cache line. We analyzed typical applications and found that a word or half-word granularity suffices, so we selected the half-word granularity for our experiments.

All lines enter the invalid state (I) upon initialization of the cache. A read miss (rm) will make the controller fetch a clean memory block, setting the line in the clean (C) state (the control bits are *don't care* terms). A write hit on this clean line will switch the state to D_v, and the control bit(s) of the written half-word(s) will be set (meaning valid and dirty). The other control bits will be cleared (meaning valid and clean). Further write hits (wh) on a line in the D_v state will set the corresponding control bits. Note that any tag hit on a line in this state is also a half-word hit because all half-words are valid, irrespective of whether their control bit is set or cleared.

A write miss (wm) will allocate a cache line by selecting a victim, but will not fetch the missing block from memory to the cache, and the line will switch to the D_{pv} state. The control bits of all half-words will be cleared, except for those half-words that are written upon after the write miss, For these updates half-words, the control bit will be set, indicating that the half-word is both dirty and valid. Further read or write hits on valid (dirty) half-words in the line ($rh(v)$) imply no action. Write hits on an invalid half-word (i.e. tag hits, half-word misses) set the corresponding control bit. A write hit on the last invalid half-word in the line ($wh(last)$) will make the state change to the D_v state with no further action. A read hit on the line, addressing a half-word which is set invalid (i.e. a tag hit, word miss, noted $rh(i)$) will also switch the state to D_v, but the controller will fetch the block from memory and will merge it in the cache line with the half-words that have their control bit set.

The rpl event in the diagram stands for evictions. When a line is evicted I becomes a transitory state leading to one of the three other states. The termination of a thread block triggers an invalidation of all the tinyCache lines in the SM where the block was executing. The *rpl* event on a line in the D_v or D_{pv} states triggers a write-back of the valid (dirty) half-words ($rpl : wb(v)$).

3.1 Maintaining coherence across all the tinyCaches

The biggest implication of having a cache per lane is the task of maintaining coherence across the lanes, since one or more lane may cache the same line from either the scratchpad memory or global memory. However the CUDA/OpenCL programming model allows us to take certain liberties to establish coherence between these tinyCaches with a lower overhead. There are five scenarios to consider:

Some lanes write to different locations in scratchpad memory/global memory: A typical programming pattern in CUDA applications is that consecutive threads reference consecutive memory elements and therefore consecutive lanes will share the same memory block in their tinyCaches, referencing only a part of it. This false sharing would wreak havoc on a coherence protocol when it comes to writings. The write-validate, write-back protocol described earlier let us obviate any invalidation or update across the tinyCaches. The CUDA/OpenCL programming model only ensures sequential consistency on writes by a single thread [6]. A thread reading a shared variable will not see a change made by another thread unless explicitly using a barrier, irrespective of whether or not these threads belong to the same warp or block, or whether they access the scratchpad memory or the global memory. Assume x(0) and x(1) are two array elements lying on the same cache block. Thread 0 writes element x(0), and then reads element x(1), whereas thread 1, which is running concurrently, writes element x(1) and then reads x(0). The model does not allow any assumption on the values read from x(1) and (x2). On the other hand, both threads will have a copy of the same memory block in their tinyCache, in D_v state, with different control (dirty) bits set. When these blocks are replaced, the dirty half-words will be correctly written-back to memory. It is precisely this subtlety that we exploit to use incoherent caches.

Some lanes write to the same location in scratchpad memory/ global memory: Since there is no ordering of threads, the programming model does not guarantee which thread will first write to the address [6]. Thus, we do not need to take further actions beyond the ones implicit in the write-validate policy with write-back as described above.

Atomic operations: Atomic operations need to guarantee a consistent view of the memory. To avoid the coherence overhead, we do not cache these atomic addresses in the tinyCache.

Synchronization primitives (barriers) : When we hit a barrier, we evict all the tinyCache entries.

A single byte write access cached by the tinyCache: Since the cache line only has a control bit per half-word, we do not cache such addresses. Effectively, we trigger an eviction if the address was already cached or just bypass the tinyCache if the address was not cached.

Local memory is transparent to the programmer. It is a specially allocated area of the global memory, used for spill code. Since there have been numerous techniques that improve the utilization and effective capacity of the register file, and given the low frequency of its occurrence, we reserve the tinyCache only for the global and shared addresses.

3.2 Area overhead

A control bit per half-word implies 32 control bits per line for a tinyCache with 64 B line size. The tag SRAM has two bits for the state plus 79-bit tag for a total of 81 bits per line. The tag is a part of the extended address that can indicate if a given reference is a global or a scratchpad memory reference, as well as the block id if needed. Thus each tinyCache line needs 15B of meta data. Therefore each 16-entry tinyCache costs us just over 1 KB (1264 B), 19% of which is control overhead.

In all, the tinyCaches account for about 9% of the area of the GPU on-chip memory hierarchy. Our estimation is based on GPUSimPow [7], including coalescing, shared memory, constant and texture cache, and the L2 cache, but excluding the LLC.

4. EXPERIMENTAL SETUP

Parameter	Configuration
SM cores	4 SMs, 1.5 Ghz
Number of lane per SM	32, in-order
Memory Coalescing	Enabled
Maximum concurrent blocks per SM	8
Maximum concurrent warps per SM	24
tinyCache per lane	1KB / 8-way / 64B line /1 cycle
scratchpad memory per SM	48KB / 8 banks / 18 cycles
L1 cache per SM	32KB / 8-way /128B line/ 18 cycles
L2 cache per die	256KB / 16-way /128B line/ 7 cycles
LLC per die	8MB / 32-way / 128B line /14 cycles
Memory	18GBytes/s BW with 50ns access time

Table 1: Simulation parameters

Our baseline architecture, depicted in Figure 2a, is based on the specifications of Fermi [4]. Following the trend of integrating GPUs and CPUs in a single die sharing caches [8], we incorporate a LLC in our baseline, similar to [8–10]. Some relevant configuration parameters are listed in Table 1. Our proposal to add a tinyCache per lane, shown in Figure 2b, is built on top of the baseline. The tinyCaches were sized and its parameters were chosen on the basis of the energy-delay product (ED), and the IPC. More details about the sizing are available in Section 5.2. We use CACTI [11] to estimate the latencies of the memory structures, except for the DL1, whose complex architecture is not modeled well with CACTI. We rely on measurements and published Fermi latencies that estimate the DL1 latency to be around 18 cycles. Our assumptions are consistent with [12] and [8].

ESESC [13] is a simulator that supports several ISAs (ARM and SPARC). We extended the simulator to use PTX instructions. ESESC execution driven simulation uses GPU instructions and simulates an array of SMs, modelling the baseline architecture (timing model of the compute units). To model a GPU, we use public information from NVIDIA Fermi [4] like the maximum number of threads that can be allocated per SM based on register file and scratchpad memory usage per thread, and the hardware limits set for an architecture. To model SIMD execution, we simulate the execution of threads in a warp by advancing the program counter in a lock-step fashion, syncing intermittently per basic block. This allows threads to diverge if needed until they reach the basic block where they finally re-converge. For warp scheduling, we model a simple round-robin, single-level mechanism, and we switch warps at memory operation boundaries. There are fixed number of threads per warp, and we limit the maximum number of in-flight (active) warps to the limit specified for a specific architecture. This is the default option in other GPGPU simulators (GPGPU-Sim [14] and Multi2Sim [15]). We model memory coalescing.

Benchmark	Description
Backprop	A machine learning algorithm used in a graph
BFS	A graph traversal algorithm
Convolution (convo)	An image processing algorithm
HotSpot	A tool to estimate the temperature for an architectural floorplan
SAXPY	A common subroutine which performs $z = \alpha * x + y$
SGEMM	Matrix Multiplication
SPMV	A commonly used implementation for multiplication of sparse matrices
SRAD	Used to remove locally correlated noise in an image
Transpose	Compute the transpose of a matrix

Table 2: GPU workloads used in our evaluation.

Our power model is based on GPUSimPow [7] and CACTI. Our estimations include only the on-chip dynamic energy, we do not include DRAM. As far as the tinyCaches go, we do not treat the line fills and coalescing requests like other regular requests; they are more expensive and accounted for separately. We expect the same leakage as with DL1 memory structures (without coalescing and crossbars), which should be less than 10% of the leakage of the total on-chip memory hierarchy.

Our benchmarks are from Rodinia [16] and Parboil [17] in addition to a few regular benchmarks from the CUDA SDK [18]. Table 2 lists and briefly describes the GPU workloads that we use in our evaluation. All benchmarks from the standard benchmark suites were run to completion with the largest dataset made available. SAXPY uses 8MB arrays. Convolution was performed on a 3072×3072 sized 2D image. Transpose uses a 2688×2688 square matrix as its input.

5. ANALYSIS AND OBSERVATIONS

Section 5.1 presents our main results and emphasizes the ability of the tinyCache to meaningfully filter out accesses to the DL1G, cutting down the energy. The sizing of the tinyCache is presented in Section 5.2.

5.1 Main Results

Different benchmarks exhibit different memory patterns, dominated by global references, scratchpad memory references or both. To maximize the potential of the limited entries in the tinyCache, each benchmark needs a specific set of references to be cached. We ran experiments where the tinyCache was allowed to cache only the global or only the scratchpad memory or both references and noted which configuration was the most effective in minimizing the energy delay product for each benchmark.

Typically programmers are encouraged to make use of the scratchpad memory, and we see that caching these references does not affect the caching ability of the tinyCache for global references. We pick this configuration as our default policy and refer to it as ***tCbase***. To highlight the potential of tinyCaches for energy savings and their impact on performance we compare *tCbase* with a handpicked optimal configuration per application ***tCpick***, which could be a configuration that cached either one or both global and scratchpad memory references, offering maximum savings in the energy-delay product. Both are normalized to the baseline with no tinyCache ***notC***.

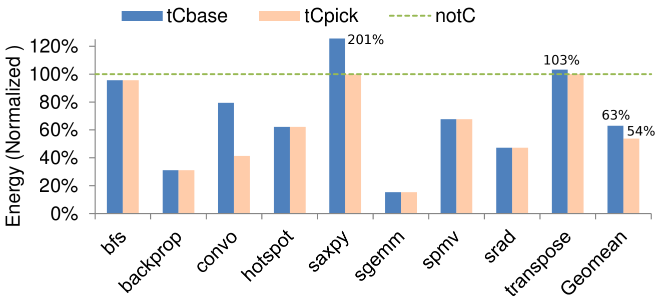

Figure 4: Addition of a tinyCache per lane reduces the dynamic energy consumption of the on-chip memory hierarchy by 37%.

A well designed tinyCache will be effective in filtering out frequent expensive requests to the DL1G and replacing these requests with energy-efficient accesses to the tinyCache. Figure 4 highlights that our base policy, *tCbase*, which caches both the global and scratchpad memory accesses, is able to achieve a significant reduction of around 37% in the total dynamic energy consumed by the on-chip memory hierarchy. Note that *tCpick* is just *tCbase* in many benchmarks, except in the case of convo and SAXPY. SAXPY performs significantly worse with *tCbase* policy. SAXPY is a streaming application with no re-use of the data it fetches or intra-thread locality. In addition, each miss on the tinyCache entails an expensive line fill operation from the DL1. Thus the presence of the tinyCache only adds overhead without filtering any traffic from DL1G. As a result, it is not beneficial to use a tinyCache with such applications. Since our emphasis for this paper is the energy efficiency, *tCpick* for SAXPY is essentially the same as *notC*, since by bypassing global, we do not cache any references in the tinyCache. We could disable the tinyCache by exposing it through the API to the driver that spawns the kernels, similar to disabling L1 caching in the Fermi. We could also potentially detect streaming accesses in hardware and disable the tinyCache transparently; we leave this exploration to future work.

Figure 5 shows that the IPC per lane does not vary significantly across different policies. This is because of the large number of

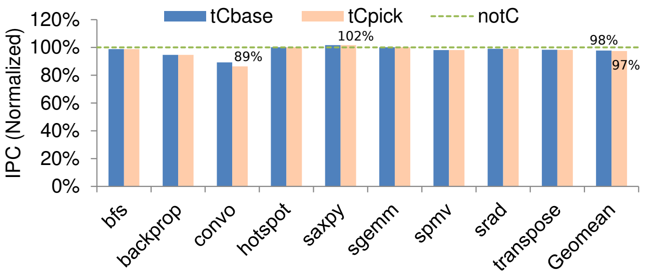

Figure 5: The IPC remains largely unchanged for most benchmarks, with a 2.3% reduction seen on average.

threads available on the GPGPU for interleaving, which makes it possible to tolerate a wide range of memory latencies. We note that unlike the increase reported in SAXPY's energy numbers, the tinyCache does not hurt its performance. Transpose is a similar case. This application is right behind SAXPY, when we consider the tinyCache miss ratio, and yet the performance is not affected. This is because the latency added by the tinyCache, very small compared with the rest of the memory levels, is easily engulfed by the warping mechanism. We see a similar decrease in the IPC, even if we opt for the *tCpick* policy.

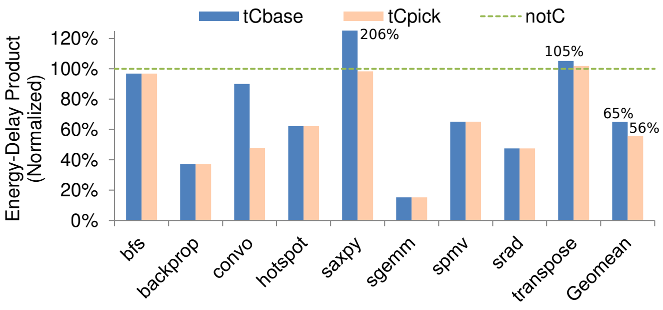

Figure 6: We see a 35% reduction in the ED after the addition of a tinyCache per lane

The ED plot shown in Figure 6 closely follows Figure 4. The average reduction in the Energy-Delay product with respect to the baseline in Figure 6 highlights how tinyCaches achieve substantial energy savings for the moderate performance losses observed in Figure 5. Applications like backprop, SGEMM, SPMV and SRAD benefit very well with a tinyCache: they show a large decrease in the energy (68%, 85%, 32% and 52% respectively) for a negligible loss in performance. Convo yields good numbers if we bypass the scratchpad memory references and provide more space for the global references, but a streaming application like SAXPY does not benefit at all.

Figure 7 shows the breakdown of memory accesses to each memory level, normalized to the baseline (no tinyCache). The leftmost bar in each group plots the baseline breakdown as a reference. The bars in the center and on the right stand for the base tinyCache (*tCbase*) and the best pick per application (*tCpick*) according to Figure 8. We observe that the number of accesses to DL1G drops by almost 62% on an average, and by almost 81% for scratchpad memory references. Note that an access to the tinyCache is significantly lesser expensive compared to DL1G. This is because an access to the tinyCache does not require going through the coalescer and the address and data distribution network, which are expensive parts in terms of energy consumption. The L2 cache shows a minor improvement as well. This is probably because the tinyCache is able to retain references that would otherwise have been displaced from the DL1G and incurred a costly L2 reference.

We should note that memory accesses that would have coalesced in a typical GPGPU memory hierarchy still have the same benefits in the tinyCache configuration, except that they may have less data reuse in the tinyCache. Effectively, all the tinyCaches have the same address cache misses and these tinyCache misses are coa-

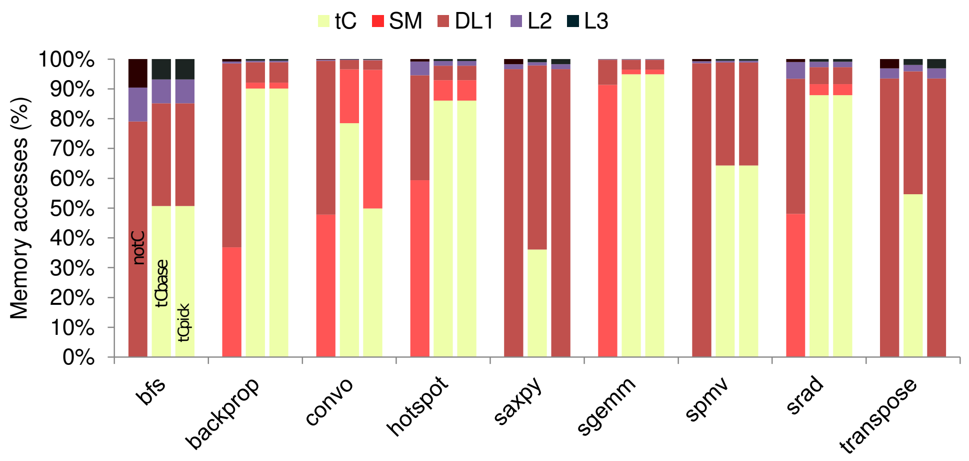

Figure 7: A breakdown of the memory accesses seen by each cache in the on-chip memory hierarchy for the baseline, the baseline tinyCache and the best pick per application. We see a 61.8% and 81% reduction to the number of accesses to the DL1G and scratchpad memory respectively.

lesced. Due to evictions at different times, we could potentially have a higher number of DRAM accesses. The worst case is evident in SAXPY, where the number of DRAM accesses nearly doubles with tinyCaches, increasing from a 1.7% global miss rate to 3%. Nevertheless, this is the worst case benchmark, and on an average we maintain the same global miss rate of 1.5% across applications.

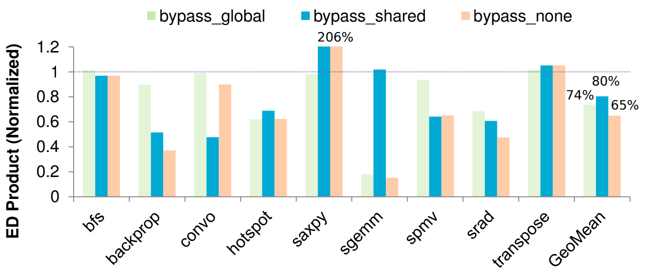

Figure 8: A tinyCache which caches both global and scratchpad memory references provides the most optimal savings in ED (and E) (**bypass_none**) as compared with a configuration caching only global references (**bypass_shared**) and one caching only scratchpad memory references (**bypass_global**).

We see that a tinyCache that caches both the global and scratchpad memory references is often the configuration that yields the best savings in E and the ED product, as seen in Figure 8. The long latencies associated with global references make them a good candidate for caching. However, this is not necessarily the optimal choice for all benchmarks. For example, an application like SGEMM which is highly scratchpad memory intensive prefers a configuration where we cache only the shared references and exploit the spatial locality exhibited. On the other hand, convo, which involves both global and scratchpad memory references, prefers caching global references only. The reason behind is how consecutive threads reference neighboring array elements in convo. When these elements fall in different cache lines some thrashing may occur. We have observed that this effect strongly decreases in convo as we increase the size of the tinyCache.

5.2 Sizing the tinyCache

Determining the size of the tinyCache is a crucial step. In the architecture we propose, we add an extra level in the memory hierarchy –tinyCaches– between the lane and the DL1G to save energy by reducing the distance traversed by on-chip data transfers as close as thread locality allows, filtering requests that otherwise pass through the coalescing logic and go to the DL1G. This may sacrifice performance on behalf of the energy savings if the hit rate in this new level is low, because all the misses incur an additional delay. The tinyCache thus needs to be large enough to retain as many memory references as possible to maximize the hit rate, but conveniently small for energy efficiency. Since we know that memory access patterns are usually quite varied, a poorly designed cache might not be able to provide the benefit we seek. Our goal in picking a configuration was primarily energy efficiency, but not at the cost of a significant performance difference. We thus use ED as our primary metric to pick the tinyCache configuration.

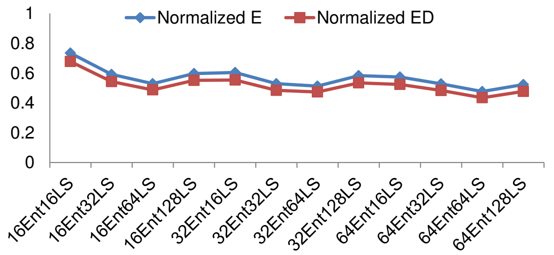

Figure 9: Impact of varying the line sizes and entries in the tinyCache on Energy (E) and Energy Delay (ED).

We observe that the IPC per lane hardly fluctuates along the design space by changing the configuration of the tinyCache. Figure 9 plots the normalized E and ED. We have chosen 16 entries and a line size of 64 B, as it is the smallest configuration with minimum ED, to keep leakage low. In any case, as mentioned in Section 5.1, the overall leakage on the tinyCaches of the 32 lanes in this range is negligible compared to the rest of the system, and will likely be compensated by the reduction in temperature on DL1 and the coalescer.

6. RELATED WORK

Most of the research on GPGPUs is steered toward improving their ability to sustain a high throughput (performance) in a discrete system. The trade-off between the thread block size, blocks per SM and storage allocation (registers and scratchpad memory) is an important issue to be understood by every CUDA programmer, and is documented in [6]. Bakhoda *et al.* [14] noted the criticality of the memory access latencies to the performance of non-graphics applications and their sensitivity to the memory bandwidth.

With the newer generations of GPGPUs getting larger on one hand, and the adoption of mobile GPGPUs on the other, there is a growing emphasis on the energy efficiency of GPGPUs.

Gebhart *et al.* explore several register allocation algorithms and propose a compiler specifiable register file hierarchy that allows sharing of temporary register file resources among running threads, reducing the usage of this energy hogging resource [19], [12]. They also propose a unified scratch, register and primary cache that can be configured at runtime to minimize the access latencies [20].

The filter cache was first proposed for uniprocessors in [3], where the authors proposed power consumption as an important and valid tradeoff for a loss in performance. The idea was extended to multicore processors in [21], where the best configuration included a victim cache at the filter cache level. A figure with a per-lane private data cache appears before the first level cache in the organization of the Echelon GPU architecture [22], but there is no mention or explanation of any kind in the write-up. To the best of our knowledge, we are the first to propose a data cache per lane, exploiting the programming model to hide the overhead of coherence.

The tinyCache can be empowered by combining them with many ideas proposed in the references above, particularly those that attempt to improve the inter and intra -thread and block level locality.

7. CONCLUSIONS

Meeting the power budget is often the biggest challenge to overcome as we move toward the next generation of parallel processors. With advancements in the programming model, the trend is toward a growing number of cores and a severely lagging memory system supporting it. As GPGPUs get more pervasive, be it through phones, or hand held gaming consoles, or cameras, energy efficiency becomes more crucial. In this paper, we propose adding a tinyCache per lane in the GPGPU to filter out requests to the energy inefficient DL1 and shared memory to save energy. We do this by exploiting features of the unique programming model and avoid incurring the overhead associated with coherence. We see a substantial saving of roughly 37% of the energy consumed by the on-chip cache hierarchy on average, and a 35% reduction in the energy-delay product. While there are some memory access patterns that can benefit with larger tinyCache capacities or by caching more types of references, the difference is not large enough to justify a complex adaptive mechanism. The tinyCache also makes it possible for us to think beyond the data access patterns typically used while writing GPGPU applications, and exploit locality in the tinyCache to gain further savings.

Acknowledgements

We would like to thank the reviewers for their feedback on our paper. Special thanks to Dr. Nuwan Jayasena for his constructive feedback. This work was supported in part by the National Science Foundation under grants 1059442 and 1205377; Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the NSF. This work was also supported in part by grants TIN2010-21291-C02-01 (Spanish Gov. and European ERDF), gaZ: T48 research group (Aragón Gov. and European ESF), and HiPEAC-2 NoE (European FP7/ICT 217068).

8. REFERENCES

[1] (2012) NVIDIA GK110. NVIDIA Corporation. Santa Clara, CA. [Online]. Available: http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

[2] (2012) AMD Graphics Core Next (GCN) Architecture. AMD, Inc. Sunnyvale, CA. [Online]. Available: http://www.amd.com/jp/Documents/GCN_Architecture_whitepaper.pdf

[3] J. Kin, M. Gupta, and W. H. Mangione-Smith, "The filter cache: an energy efficient memory structure," in *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, ser. MICRO 30. Washington, DC, USA: IEEE Computer Society, 1997, pp. 184–193.

[4] (2010) NVIDIA GF100. NVIDIA Corporation. Santa Clara, CA. [Online]. Available: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

[5] N. P. Jouppi, "Cache write policies and performance," in *Proceedings of the 20th annual international symposium on computer architecture*, ser. ISCA '93. New York, NY, USA: ACM, 1993, pp. 191–201.

[6] *NVIDIA CUDA C Programming Manual*, NVIDIA Corporation, Santa Clara, CA, 2012.

[7] J. Lucas, S. Lal, M. Andersch, M. Alvarez-Mesa, and B. Juurlink, "How a Single Chip Causes Massive Power Bills GPUSimPow: A GPGPU Power Simulator," in *IEEE International Symposium on Performance Analysis of Systems and Software*, 2013.

[8] M. Yuffe, E. Knoll, M. Mehalel, J. Shor, and T. Kurts, "A fully integrated multi-cpu, gpu and memory controller 32nm processor," in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2011 IEEE International*, feb. 2011, pp. 264 –266.

[9] Y. Yang, P. Xiang, M. Mantor, and H. Zhou, "Cpu-assisted gpgpu on fused cpu-gpu architectures," in *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture*, ser. HPCA '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 1–12. [Online]. Available: http://dx.doi.org/10.1109/HPCA.2012.6168948

[10] J. Lee and H. Kim, "Tap: A tlp-aware cache management policy for a cpu-gpu heterogeneous architecture," in *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, feb. 2012, pp. 1 –12.

[11] S. Wilton and N. Jouppi, "CACTI: An Enhanced Cache Access and Cycle Time Model," *IEEE Journal of Solid-State Circuits*, vol. 31, no. 5, pp. 677–688, May 1996.

[12] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron, "Energy-efficient mechanisms for managing thread context in throughput processors," in *Proceedings of the 38th annual international symposium on Computer architecture*, ser. ISCA '11. New York, NY, USA: ACM, 2011, pp. 235–246.

[13] E. K. Ardestani and J. Renau, "ESESC: A Fast Multicore Simulator Using Time-Based Sampling," in *International Symposium on High Performance Computer Architecture*, ser. HPCA'19, 2013.

[14] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2009)*, April 2009, pp. 163–174.

[15] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, "Multi2sim: a simulation framework for cpu-gpu computing," in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, ser. PACT '12. New York, NY, USA: ACM, 2012, pp. 335–344. [Online]. Available: http://doi.acm.org/10.1145/2370816.2370865

[16] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, ser. IISWC '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 44–54.

[17] J. Stratton, C. Rodrigues, I. Sung, N. Obeid, L. Chang, N. Anssari, G. Liu, and W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," *Center for Reliable and High-Performance Computing*, 2012.

[18] N. Corporation, "Nvidia sdk." [Online]. Available: http://developer.nvidia.com/cuda-toolkit

[19] M. Gebhart, S. W. Keckler, and W. J. Dally, "A compile-time managed multi-level register file hierarchy," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44 '11. New York, NY, USA: ACM, 2011, pp. 465–476.

[20] M. Gebhart, S. W. Keckler, B. Khailany, R. Krashinsky, and W. J. Dally, "Unifying primary cache, scratch, and register file memories in a throughput processor," in *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-45 '12, 2012.

[21] Y. J. Park, H. J. Choi, C. H. Kim, and J.-M. Kim, "Energy-aware filter cache architecture for multicore processors," in *Electronic Design, Test and Application, 2010. DELTA '10. Fifth IEEE International Symposium on*, jan. 2010, pp. 58 –62.

[22] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco, "Gpus and the future of parallel computing," *IEEE Micro*, vol. 31, no. 5, pp. 7–17, Sept. 2011.