2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing

A quantitative performance evaluation of fast on-chip memories of GPUs

Elias Konstantinidis
Department of Informatics and Telecommunications,
University of Athens,
Panepistimiopolis, 15784, Athens, Greece
Email: ekondis@di.uoa.gr

Yiannis Cotronis
Department of Informatics and Telecommunications,
University of Athens,
Panepistimiopolis, 15784, Athens, Greece
Email: cotronis@di.uoa.gr

***Abstract*—Modern Graphics Processing Units (GPUs) have evolved to high performance general purpose processors, forming an alternative to CPUs. However, programming them effectively has proven to be a challenge, not only due to the mandatory requirement of extracting massive fine grained parallelism but also due to its susceptible performance on memory traffic. Apart from regular memory caches, GPUs feature other types of fast memories as well, for instance scratchpads, texture caches, etc. In order to gain more insight to the efficient usage of these memory types some quantitative performance measures could be beneficial.**

In this paper we describe a set of micro-benchmarks which aim to provide effective bandwidth performance measurements of the on-chip special memories of GPUs. We compare the peak measurements of different memory types and the use of different data type sizes. In addition, we validate the peak measurements on real world problems as provided by the polybench-gpu benchmark suite. We compare the profiling bandwidth of on-chip memories with the peak measurements as captured with the proposed micro-benchmarks. The source code of the micro-benchmark suite is publicly available.

I. Introduction

During the last decade GPUs have emerged as an alternative option in the high performance computing sector. This is mostly due to their high performance in compute operations and their great power efficiency. In order to exploit their compute potential, programming environments like CUDA and OpenCL have been devised and evolved to enable leveraging GPUs in compute tasks.

Programming GPUs has proven to be more challenging than programming CPUs. The GPU itself is not as highly equipped with large memory caches as the CPUs do. The largest shared level cache of GPUs known at the time of writing this paper was 3MB where CPUs can be equipped with well more than 10MB of shared level cache. The CPU is able to alleviate DRAM accesses not characterized by high locality, whereas the GPU is not optimized to cope with such cases. Furthermore, GPUs are designed to execute a huge amount of threads concurrently in order to hide memory access latencies. This fact limits cache memories use mostly for spacial locality accesses. As a result many GPU kernels regarding scientific problems prove to be memory bound, e.g. sparse matrix-vector multiplication (SpMV)[19], stencil computations[2], [6], etc.

As GPUs originate from the graphics acceleration domain they carry other fast on-chip memories which are not evident on CPUs. These special memories are designed and optimized each for a different type of use. For instance, texture memory is optimized for accesses with high degree of 2D locality, whereas constant memory performs best in cases where a particular data element is broadcasted to many threads and shared memory works as a software managed cache. In addition, these special memory types cannot be used transparently just as regular caches do. Programmer has to use different constructs in order to exploit their benefits. This poses a challenge in programming GPUs.

In this paper a benchmark suite consisting of 3 micro-benchmarks is described which aims to assess the performance of fast on-chip memories as supplied by the GPUs. Each micro-benchmark focuses on the evaluation of peak effective memory bandwidth of a particular memory type. Yielded results provide hints to the programmer on which type of memory can provide highest bandwidth for a particular case. In addition, different data size types (32, 64, 128 bit) were compared regarding the attained performance.

The rest of this paper is structured as follows. In the next section the GPU memory hierarchy is described with focus on the on-chip fast memories. In section 3 the developed GPU micro-benchmarks are presented. In section 4 the micro-benchmark execution results on 3 GPUs are presented and discussed. In section 5 the execution of a third party benchmark suite is examined in order to compare the highest peak bandwidth of the most utilized on-chip memory type with the respective micro-benchmarked bandwidth results. Related work is referred in section 6 and the conclusions and future work follow in section 7.

II. GPU memory hierarchy

Typically, CPUs are based on a hierarchical cache model which consists of three to four levels of cache memory. Programs are able to transparently exploit these memories though not always as efficiently as possible, especially if the programer ignores their existence. The GPUs instead feature a greater variety of memory types with different characteristics. In a GPU kernel, beyond using global memory (GPU external DRAM) which serves as the counterpart of CPU

978-1-4673-8776-7/16 $31.00 © 2016 IEEE
DOI 10.1109/PDP.2016.56

main memory, programmers have at their disposal memory types as texture memory, shared memory or constant memory. Though, a kernel could be developed by utilizing only global memory, exploiting one or more of the later memory types could potentially offer performance benefits. Each of the aforementioned memory types has special features and fits to particular cases. However, it is up to the programmer to choose the most suitable type of memory for each case.

From the GPU programmer's perspective, the memory types that CUDA API provides are:

- **Global memory** is the most common memory type of the GPU as it is the only that can be read and written by both the host and the GPU. Physically, it is a globally visible memory space by all SMs. It resembles the regular CPU memory as modern GPUs provide caching mechanisms in 1-2 levels of caches in the same fashion as the CPUs do. The L1 cache is typically internal to each SM (Streaming Multiprocessor using CUDA terminology) whereas the L2 cache is globally shared to all SMs. L2 cache facilitates to mitigate the bandwidth pressure to device memory. It's size varies from some hundreds of KB to 3 MB. Depending on the architecture, there are cases in which global memory accesses are cached in L1 cache and others in which global memory accesses are only cached in L2. Current GPUs do not support caching of memory stores in L1 cache as L1 caches are not coherent between SMs. Experiments have shown that global memory stores are directly transferred to the L2/device memory, bypassing the L1 cache instead of following a write-back policy. The global memory on its own is very slow as it needs hundreds of cycles in order to be accessed. However, in contrast to the CPU main memory it is optimized for bandwidth and can offer more than 300 GB/sec on high end GPUs by keeping many in-flight memory transactions in it's pipeline.
- **Texture memory** offers a different path for reading global memory. With texture memory the global memory can be accessed more efficiently in cases where 2D locality is evident. In this respect each multiprocessor is equipped with a separate texture cache in order to exploit access locality. The available cache working set per multiprocessor for texture memory ranges between 12KB and 48KB (table 12 in [12]). Texture memory is read only and it is not coherent in case the memory region is updated in between the kernel execution. For memory regions accessed in streaming fashion texture memory does not offer performance benefits.
- **Shared memory** is a scratchpad, a fast memory space private to CUDA thread blocks and distinct from the global memory space. Thus, it is not a regular cache but because it is very fast with low latency, it is considered a software managed cache. Each SM is equipped with an on-chip private shared memory.
- **Constant memory** is a predefined part of global memory space which is set to be accessed as a read only memory. In this respect it resembles texture memory. However, it is clearly optimized for broadcast accesses on a thread warp basis. In order to provide fast access, each SM is equipped with a constant memory cache. The constant memory cache size is 8KB to 10KB per multiprocessor[12].

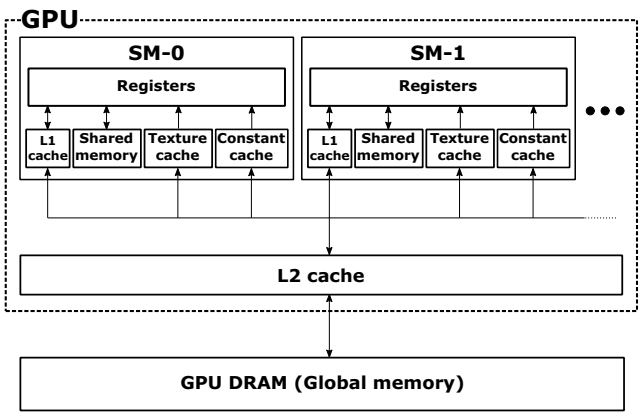

Fig. 1. GPU memory hierarchy includes caches and scratchpads.

TABLE I
THE GPU ON-CHIP MEMORY TYPES AS PROVIDED BY MODERN GPUS

Memory type	GPU physical memory storage		
	Part of the SM	Scratchpad	Read-only
Shared memory	✓	✓	
L1 cache	✓		✓*
L2 cache			
Texture cache	✓		✓
Constant cache	✓		✓

*GPUs currently do not cache global memory stores in the L1 cache though values of spilled registers and local arrays can be cached.

A diagram with GPU cache memory hierarchy is depicted on figure 1. GPU memory type characteristics are summarized on table I. Other vendors provide other special memory types as well, e.g. last generations of AMD GPUs include a fast scratchpad shared between all compute units on the GPU (Global Data Share, GDS[1]).

This variety poses a challenge for GPU programmers in order to make optimal use of the GPU memory hierarchy. Each memory type features different characteristics and it is not always clear which combination is optimum for a particular purpose. Additionally, each GPU generation brings significant architectural changes to the memory type features which potentially affect performance. Vendors do provide some information about the architectural details of the memory subsystem but they do not always present bandwidth figures.

Though, GPU global memory provides a much higher bandwidth than CPU main memory, it is of great significance to be used efficiently. In order to sustain high throughput, the streaming processors of the GPU require a vast amount of input data to consume. Therefore, GPUs cannot be fed with data at the required rate using slow global memory only. Cache memories are very small and the large amount of active threads

only makes it more difficult to alleviate pressure on global memory. If the streaming processors cannot be fed adequately, overall performance drops and gets bound to global memory bandwidth. Therefore, it is important to exploit locality with on-chip fast memories as efficiently as possible.

III. GPU MEMORY MICRO-BENCHMARKS

All micro-benchmarks developed in this work follow the approach of performing a vast amount of memory accesses while minimizing overheads. Optimizations include the use of template variables and loop unrolling where beneficial. The access strides are fixed during compilation time, thus the generated ISA code gets more optimized as the resulting load/store instructions include a source/destination address register and a fixed offset. In this case the index offsets are known during compilation time and thus no address calculation instructions are required, so the instruction mix gets more efficient. Thus, no redundant instructions as address calculations are generated (e.g. ISA instruction: *LD.E R4, [R6+0x1800]*). It is necessary to eliminate as much as possible the extra overhead caused by instructions not being load/store operations in order to reliably estimate bandwidth. All kernels are configured to run by accessing configurable element sizes i.e. 32bit (int), 64bit (int2) or 128bit (int4). Optimum performance is most likely provided by using such native types instead of using other data type widths, e.g. 96bit or 192bit. In this respect the impact of memory performance by using different element sizes could be explored. All micro-benchmark source codes are freely available for experimentation[1] under the GNU GPL 2 license.

A. cachebench (L1, L2 & texture cache micro-benchmark)

Benchmarking the multilevel cache hierarchy is achieved by a kernel in which threads of a warp access elements repeatedly in a sequence using strides equal to thread block size (figure 2). Strides are applied for a predefined number of iterations (threshold) which is set at the compilation time and it virtually determines the size of the dataset that is being accessed. Afterwards, the index is reset to its initial position and the procedure continues repeatedly. For smaller datasets than the size of grid of threads some of the indexes of different threads are overlapped. Each thread performs a large total number of accesses (8192) in order to eliminate the relative extra overhead of other instructions and an adequate amount of thread blocks is created at the invocation in order to keep the GPU device occupied (SMs $\times$ Max Resident Blocks per SM). A few experiments are conducted in which the threshold is progressively increased. The larger the threshold is, the bigger the dataset that is intensively accessed. In order to trick the compiler so it is hindered to eliminate any memory accesses, a fake update to the index of the array is performed. A bitwise XOR operation is applied to the index/element pointer with the data that was read from memory but as long as the dataset has been initialized with zero values no change to the index is actually made.

[1]https://github.com/ekondis/gpumembench

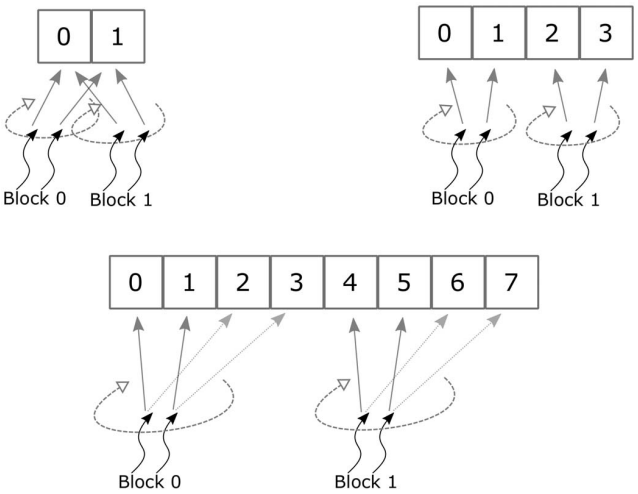

Fig. 2. Thread accesses in the cachebench micro-benchmark under a simplified scenario with 2 thread blocks, 2 threads per block and 3 different configurations of the dataset (2, 4 and 8 elements). Each element is accessed multiple times.

Two different configurations of the kernel are built. The first performs only read operations on the dataset elements whereas the second does a copy of the dataset elements from one area of the array to another by writing the element values that it reads on a separate region of the array. Streaming stores are used in order to limit cache pollution (PTX *"cs"* cache streaming operator[13]).

The kernel is implemented by either reading elements directly from global memory or by reading elements via texture memory. So both types of caches, L1 cache and texture cache, can be assessed. In case the elements are read directly from global memory, two executables are generated, one with all cache levels enabled (PTX *"ca"* cache operator[13]) and one with L1 cache disabled (PTX *"cg"* cache operator[13]). The purpose of having two different configurations is to examine the L2 cache behavior in isolation of the L1 cache effects.

Summarizing, three executables are generated which differ by the type of memory loads:

1) Global memory with all caches enabled
2) Global memory using L2 cache only
3) Texture memory

B. shmembench (shared memory micro-benchmark)

The shared memory micro-benchmark works by exchanging repeatedly scalar/vector values in shared memory. Each swap amounts to two load and two store accesses. Thread synchronization barriers are also limited to *__threadfence_block()* as the primary goal in the kernel is to evaluate the throughput of shared memory accesses through data exchanges and not to involve extra synchronization overheads. Element accesses are sequential between threads in a warp thus no bank conflicts occur. The default shared memory configuration has been used (*cudaSharedMemBankSizeDefault*). Each thread performs a total of $5 \times 1024 = 5K$ element swaps in shared memory.

TABLE II
HARDWARE SPECIFICATIONS OF THE GPUS USED IN THE EXPERIMENTS

GPU (CC[1])	L1 cache (KB per SM)	Texture cache (KB per SM)	Shared memory (KB per SM)	Constant memory cache (KB per SM)	L2 cache (KB per GPU)
GTX-480 (CC2.0)	16/48[2]	12	16/48[2]	8	768
GTX-660 (CC3.0)	16/32/48[2]	12-48	16/32/48[2]	8	384
GTX-960 (CC5.2)	24[3]	24[3]	96	10	1,024

[1] Compute Capability
[2] 64KB split into L1 cache plus shared memory
[3] Unified 24KB L1/texture cache

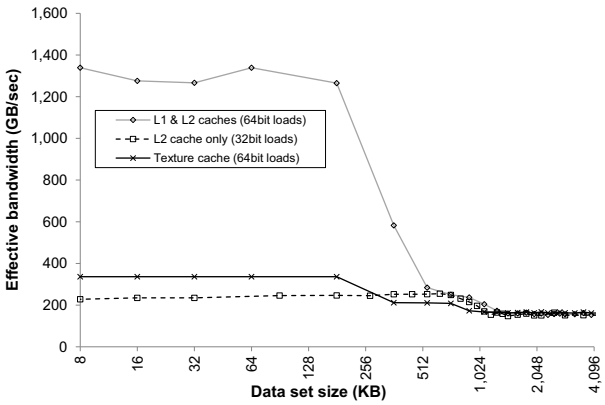

Fig. 3. Evaluation of L1, L2 & texture caches on the GTX-480.

C. constbench (constant cache micro-benchmark)

The estimation of constant memory bandwidth relies on a large number of threads which perform exactly the same memory loads in a sequential pattern in a constant memory region. The constant memory dataset consists of 1024 elements and every thread reads all elements and calculates their sum. As all threads in a warp access the same elements per time-step the constant memory cache broadcasts the element value to all threads of the thread warp which is the intended use of constant memory. The calculation is performed in multiple iterations and all threads perform the same computation.

IV. BENCHMARK EXECUTION RESULTS

We executed experiments on 3 different NVidia GPUs each having a different Compute Capability (CC) and representing a different architecture. The GTX-480, GTX-660 and GTX-960 correspond to the Fermi, Kepler and Maxwell GPU architectures, respectively. The different CC value entails differentiations on some of the features of the embedded fast memories of the GPU. The characteristics of fast memories of all tested GPUs are summarized in table II [12], [11]. Computer systems were operating on 64bit Linux OS with CUDA versions ranging from v5.5 to v7.0 (GTX-480:v5.5 and GTX-660 & GTX-960:v7.0).

A. cachebench

The purpose of this benchmark is to probe on the performance of L1, L2 and texture caches. In the conducted experiments the execution dataset ranged from 2KB up to 23MB. In particular, the maximum dataset size on the experiment was a function of the amount of the launched thread blocks, which was set to be a multiple of the amount of SMs available ($\frac{maxThreadsPerMultiProcessor}{BlockSize} \times multiProcessorCount$).

In addition, the L1 cache/shared memory configuration was set to default, i.e. the L1 cache per SM for Fermi (GTX-480) and Kepler (GTX-660) based GPUs was set to 16KB.

The results presented in this section regard the execution of the micro-benchmark using memory loads only. The reason of this is that the memory copy version which produces both load and store operations exhibited write-through behavior for the memory store operations which turned out to be the bottleneck for very small datasets. Global memory stores always generated store transactions on L2 cache regardless of data being cached in L1 cache or not. Therefore using the memory copy micro-benchmark did not allow the L1 and texture caches expose their full potential. In figures 3, 4 and 5 the results of executions of the memory load micro-benchmark are depicted for each GPU respectively. Each experiment was conducted with either 32, 64 or 128 bit data types and thus, we chose to illustrate only the best performing results.

On the GTX-480 the L1 cache exhibits the best performance (figure 3). Texture cache provided a bandwidth improvement up to ≈336GB/sec. The L2 cache aids in increasing bandwidth (≈210GB/sec) until it drops to the DRAM sustained bandwidth (≈160GB/sec). In general, the L1 cache offers great performance for small high locality datasets (≤180KB).

The GTX-660, though does not support global memory caching in L1 cache, exhibits significant improvements by utilizing texture caching (≈717GB/sec, figure 4). The *L2 only* and *L1 & L2* lines overlap as the two experiments essentially do not differ for this GPU. Relying only on global memory caching provides small bandwidth improvement (≈178GB/sec) until it converges to the sustained DRAM bandwidth (≈117GB/sec).

The Maxwell architecture based GPU, the GTX-960, features unified L1 and texture caches and thus, the L1 and texture experiments exhibited similar behavior (figure 5). The effective bandwidth reached to ≈718GB/sec on small datasets, stressed to ≈280GB/sec when L2 caching utilization was high and finally dropped to ≈89GB/sec by having accesses served by DRAM. Maxwell architecture is known to feature texture compression. However, no notable improvement in the effective bandwidth was observed in our experiments, even though the data was initialized with zeros. We believe that texture compression is restricted to the use of graphics APIs.

On the overall all data types (32/64/128 bit) provided similar bandwidth with an exception on texture memory. Texture memory did not provide near to peak performance by using 32bit data types (int). The 32bit data type exhibited peak rates 168GB/sec, 318GB/sec and 358GB/sec on GTX-480, GTX-

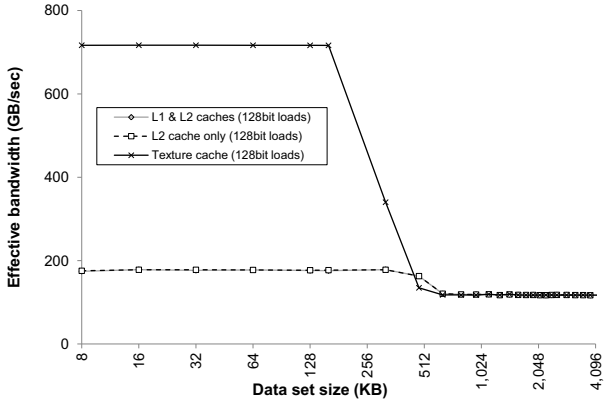

Fig. 4. Evaluation of L1, L2 & texture caches on the GTX-660.

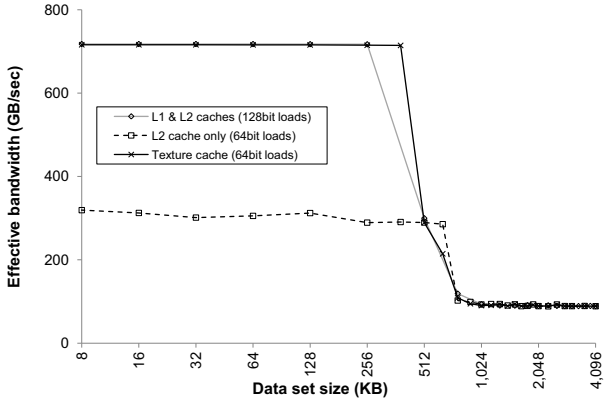

Fig. 5. Evaluation of L1, L2 & texture caches on the GTX-960.

660 and GTX-960, respectively. These rates constitute just the 50%, 44% or 50% of the rated peak with larger data types (64 or 128 bit). After performing profiling on the texture transactions and the texture cache hit rate (*tex_cache_transactions* and *tex_cache_hit_rate* profiling metrics) it was found that accessing an array sequentially by using 32bit data types required the double amount of texture transactions than using a 64bit data type. Moreover, the texture cache hit rate approached to 50% by using using 32bit type, whereas it remained 0% by using 64bit data type. Therefore, it is clear that the 32bit data type leads to the double amount of data to be fetched for each transaction. The unused part of the fetched data remained in cache and used by a subsequent texture fetch, consequently leading to a 50% cache hit ratio even if seemingly there is no data reuse. As 32bit loads per thread correspond to 128 byte loads per thread warp ($32/4 \times 32$), a minimum texture fetch transaction seems to be equal to 256 bytes. In high locality cases this seems to affect performance and in such cases a wider data type is recommended for texture fetching (64 or 128 bit). Profiling also revealed that texture cache transaction count (*tex_cache_transactions*), L2 cache transaction counts (*l2_read_transactions* and *l2_write_transactions*)

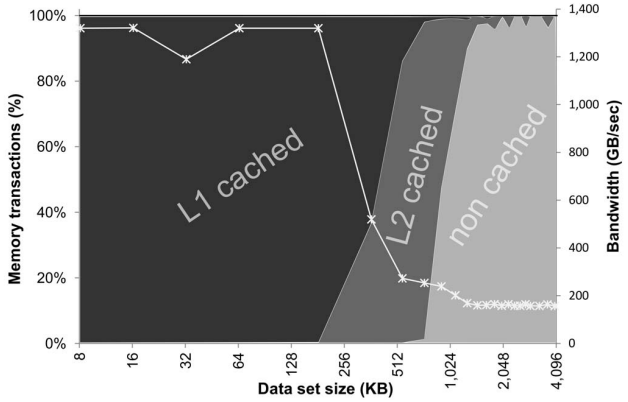

Fig. 6. Memory traffic dissection to L1 (hits), L2 (hits) & non cached accesses on the GTX-480 using global memory with 64bit elements.

and DRAM transaction counts (*dram_read_transactions* and *dram_write_transactions*) correspond to 32 bytes which is consistent with the vendor's programming guide[12].

In order to further validate the micro-benchmark tool a dissection of the induced memory traffic was conducted under execution. For each dataset size the total memory traffic was classified according to the type of memory that serviced each transaction. The aforementioned profiling metrics were used, plus the L1 cache hit counter (*l1_global_load_hit*) in order to deduce the percentage of memory traffic that was serviced by each resource i.e. traffic serviced by hits in the L1 cache, hits in the L2 cache or in DRAM. For the GTX-480 GPU the results are illustrated on figure 6 (64bit data type). It is clear that the effective bandwidth is determined by the percentage of hits in either type of cache.

B. shmembench & constbench

The shared memory and constant memory micro-benchmarks were used to assess the maximum bandwidth of both memories. The results of executions for both benchmarks are depicted in figure 7. Note that in the constant memory benchmark all threads in a thread warp read the same element which is a value being broadcasted to all threads. This broadcast in a thread warp was accounted as a set of memory loads equal to the size of the data type multiplied by the count of threads in the warp (i.e. 32), regardless of accessing the same value. Ultimately, that is the proper usage of constant memory.

The GTX-480 saturates shared memory bandwidth yielding ≈1,341GB/sec, by using 32bit/64bit elements. Constant memory performance approached 2TB/sec by using 64bit vectors.

The peak bandwidth of the GTX-660, for shared memory was ≈1,434GB/sec and for constant memory ≈1,812GB/sec by using 64bit elements in both cases. In addition, the performance in both cases when using 32bit elements was quite poor as it was less than half of the peak. Therefore, it is recommended to use at least 64bit types on Kepler GPUs.

The GTX-960 exhibited best performance in the constant memory micro-benchmark by using 64bit elements as it

TABLE III
PEAK MEASURED BANDWIDTH WITH EACH MEMORY TYPE ON ALL GPUS. VALUES IN BRACKETS ARE THE NORMALIZED BANDWIDTH RATES IN 32BIT ELEMENTS PER SM, PER CLOCK.

GPU	Effective memory bandwidth (GB/sec) - (ints/SM/clock)				
	L1 cache GB/sec	Texture cache	Shared mem.	Constant memory cache	L2 loads/ L2 copy/ DRAM
GTX-480	1,341	336	1,340	1,990	255/394/164
	(16.0)	(4.0)	(15.9)	(23.7)	(3.0/4.7/2.0)
GTX-660	178	717	1,434	1,812	178/306/119
	(8.1)	(32.7)	(65.4)	(82.6)	(8.1/13.9/5.4)
GTX-960	718	718	1,285	1,806	319/511/89
	(16.9)	(16.9)	(30.2)	(42.5)	(7.5/12.0/2.1)

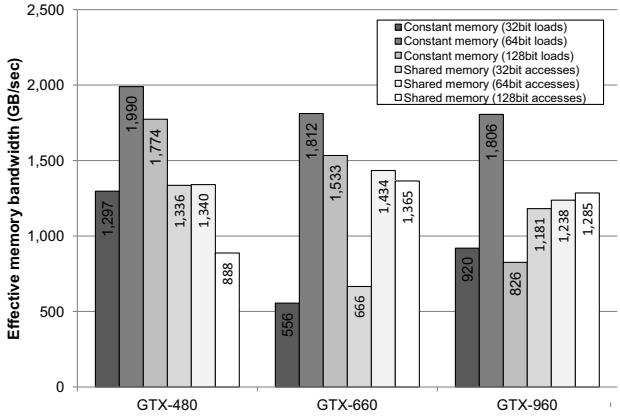

Fig. 7. Constant and shared effective memory bandwidth as measured on all tested GPUs with 32/64/128 bit data types.

reached to ≈1,806GB/sec. Shared memory bandwidth was balanced in all three cases and reached to ≈1,285GB/sec.

In most cases it is clear that the 32bit case does not perform as optimally as the 64bit or 128bit cases. In particular, constant memory in the overall performs better with 64bit elements.

In summary, the peak measured bandwidth rates of each type of memory are summarized in table III. These results combined with the respective memory sizes could provide guidance for optimizations. While the memory load micro-benchmark was used for the intra-SM memories, both types were applied to assess the L2 cache. This proved that the efficiency of the L2 cache could not be maximized by using only memory loads. It was necessary to apply both loads and stores for the L2 cache to reach peak bandwidth. Therefore, both peak values are provided on table, first by using global memory loads and second by using global memory copy. The third value is the estimated DRAM bandwidth. The numbers in brackets correspond to the normalized bandwidth in 32bit elements (ints), per SM, per GPU clock. The results regarding shared memory are consistent to the throughput of load/store units each SM provides. For instance, the GTX-480 features 16 load/store units per SM and thus, the shared memory bandwidth in 32bit elements per SM, per clock approximates the amount of load/store units per SM. The Kepler GPUs provide shared memory bandwidth of 64bits per bank, per cycle for each of the 32 banks in the SM, which is also consistent to our results. The Maxwell GPUs provide the half bandwidth per SM, i.e. 32bits per bank, per cycle. Regarding the constant memory, its bandwidth is even higher, exceeding the capabilities of load/store units. After the produced ISA code of the micro-benchmark was investigated it was found that constant memory accesses did not require explicit load instructions in order to fetch constant data but references to constant memory were embedded into the instruction performing the operation, e.g. *IADD R0, R0, c[0x2][0xc0]*. This fact proves constant memory accesses to be quite effective as they do not occupy the load/store units.

V. EXPERIMENTS ON BENCHMARK SUITE

In order to demonstrate the bandwidth limits of memories in real world problems the *polybench-gpu* benchmark suite was employed[3]. It consists of a set of 15 benchmarks for GPUs in both CUDA and OpenCL environments. We selected a subset of 5 benchmarks which exhibited the highest rate of utilization of the L2 cache on the GTX-480. These benchmarks are likely expected to provide a high degree of locality in their kernels in order to investigate bottlenecks of fast memories. The utilization is assessed by running an event counter profiling[14] with *nvprof* to all benchmarks and measuring the *l2_utilization* metric. After profiling was conducted on the GTX-480, we chose the 5 topmost L2 utilization benchmarks (FDTD-2D, GEMM, 3MM, SYR2K and SYRK). These benchmarks consist of 9 distinct kernels.

In its original form all benchmarks in the suite were fairly straightforward. Kernels were implemented by using only global memory accesses and without any kind of special memory types. Thus, for many GPUs, part of fast memories remain idle during execution. In order to experiment with fast memory types, the 5 selected benchmarks were extended to utilize texture memory. Two variations were implemented, one using scalar texture elements and one using 4 elements texture vectors. We did not implement shared memory variations due to time constraints. The use of constant memory was not applicable to the selected kernels due to its inherent constraints. Note that the problem size in SYR2K had been changed from $2K \times 2K$ to $1K \times 1K$ in order to allow successful event profiling. Otherwise, event counter overflows during profiling prevented its successful completion.

All benchmark implementations were executed via profiling. The utilization metrics were assessed and the highest rated metric was selected in order to characterize the kernel's hotspot on each GPU. Next, the memory bandwidth on the particular resource indicated by the metric was evaluated using event counters. The L2 transactions were estimated by using the *l2_read_transactions* and *l2_write_transactions* metrics. For texture memory transactions the *tex_cache_transactions* metric was used and for the DRAM transactions the *dram_read_transactions* and *dram_write_transactions* metrics were used. Memory traffic is expressed by the amount of

transactions multiplied by 32. The memory bandwidth is estimated by dividing the memory traffic by the execution time. The ratio of the memory bandwidth to the peak measured bandwidth on the same memory type as recorded through micro-benchmarks is computed in order to compare it to the peak measured as done with the micro-benchmarks. All collected results are presented on table IV.

In the majority of cases the dominant bottleneck is either DRAM or L2 cache. In case the DRAM exhibits high utilization ($>= 9$), the percentage ratio of DRAM bandwidth to measured peak gets equally high, ranging from 77% to just over 100%. In case the L2 is highly rated, two distinct ratio values are estimated. On the first ratio the peak measured bandwidth of memory loads is considered, whereas on the second the peak measured bandwidth of memory copy is considered. Thus, depending on whether the kernel performs memory loads or balanced transactions (loads & stores), one should consider the first ratio or the second. On the overall, L2 bandwidth was rated between the memory load peak and the memory copy peak measurements. However, on the GTX-660 there were 3 cases where the L2 bandwidth exceeded the highest measured peak with the micro-benchmark (355GB/sec, 374GB/sec & 375GB/sec $\geq$ 306GB/sec). Probably, the GTX-660 requires a different balance of load/store operations or more in-flight memory operations per thread than the mix employed with the micro-benchmark in order to reach peak, but in any case this assumption requires a further investigation.

The texture cache is the dominating utilization metric in 8 cases. The corresponding ratio for the GTX-480 matches the peak bandwidth as measured with the micro-benchmark. Texture memory shows to emerge as a bottleneck on the GTX-960 GPU, in case of GEMM and 3MM benchmarks. Bandwidth was lower than measured peak (475GB/sec, $\approx {}^{2}/_{3}$ of peak) but the respective utilization was not the highest possible either (8). Unfortunately, the L1 cache is not the dominant utilization resource in any case. Note though that bandwidth derived from profiling can be different from effective bandwidth.

In summary, the resource utilization highlighted the cases where a specific memory resource was identified as a performance hotspot. In these cases our peak measurement served as an peak goal of the stressed memory resource. Once the top utilized resource is identified, performance can be more reliably predicted for a particular kernel on a target GPU. The case of the L2 cache as a bottleneck proved to be more complicated as its maximum peak is dependent on the access type mix and in particular the balance of loads/stores.

VI. Related work

Early works that shed light on the full memory subsystem of the Tesla GPU architecture were done by Wong *et al.*[18] and Volkov *et al.*[16]. The tested GPU, however, though popular at the time, did not include any global memory caches. Afterwards, the developed micro-benchmarks were applied by Wodniok *et al.* on a contemporary GPU (GTX-680)[17]. In [9] Mei *et al.* developed a set of micro-benchmarks in order to extract low level details of the Fermi & Kepler GPU architectures, e.g. L1 & L2 sizes & latencies, set associative mappings and TLB characteristics. The same information with the exception of TLB data was collected by Meltzer *et al.*[10] on the Fermi based Tesla C2070 GPU.

Other research Work is focused on optimizing cache usage. Huangfu *et al.* suggests using a profiling method in order to determine which data accesses are best to be cached in L1 cache[4]. Jia *et al.* propose a compilation time algorithm that enables the configuration of cache (on/off) by the classification of memory access patterns[5]. Towards the same goal, Li *et al.* provide a compilation time framework in order to optimally split accesses to cached and non-cached[8]. They employ GPU simulation in order to analyze memory accesses. The work of Tian *et al.* lies in the same direction as they propose the design of cache memories that selectively bypass caching[15].

Previous work regarding GPU micro-benchmarking mostly focuses on low level features of fast memories, e.g. access latencies, TLB characteristics, banks, etc. To our knowledge this is the first work focusing on pure effective bandwidth of fast memories and how this relates to the element size. The derived bandwidth bounds could potentially be used on GPU performance models that require bandwidth measurements as parameters[7]. In addition, they provide some evidence on performance of fast memories aiming at kernel optimizations.

VII. Conclusions

In this work we developed and presented a set of micro-benchmarks to assess performance of fast on-chip memories in terms of effective memory bandwidth. The GPU, apart from L1 & L2 caches, features special fast memory types which cannot be used transparently by the kernel programmer. Each distinct memory type has different features, normally fits to particular cases with different pattern/locality characteristics and its type of use is documented by vendor. However, no deep performance measures in terms of bandwidth are provided. In order to fill this gap, we provide a set of micro-benchmarks that aid to assess their performance by measuring the maximum attained effective bandwidth. Using the obtained results can be beneficial in particular cases when the kernel programmer focuses on kernel code optimization.

Experiments on 3 GPUs were run by applying the micro-benchmarks and the generated results were analyzed. The attained bandwidth behavior of various fast memory types was discussed and specific optimization guidelines were proposed. The performance of the micro-benchmark suite was also verified using memory transaction profiling. In addition, the peak bandwidth measurements were compared with profiling bandwidth on the polybench-gpu benchmark suite. The resource with the highest utilization was identified by profiling 9 different kernels. The bandwidth estimation of each memory type via profiling proved to exceed 75% of the measured peak bandwidth with the micro-benchmarks for cases where the utilization of the particular resource was *"$\geq$High (9)"*.

However, each memory type has different characteristics and the micro-benchmark measurements determine by no means the memory type that fits for a particular purpose. It is

TABLE IV
KERNEL EXECUTION RESULTS OF THE 5 SELECTED POLYBENCH-GPU BENCHMARKS ON THREE GPUS.

kernel (benchmark)	type[1]	GTX-480				GTX-660				GTX-960			
		execution time (msecs)	dominant utilization metric	resource band-width (GB/sec)[2]	% of measured peak[3]	execution time (msecs)	dominant utilization metric	resource band-width (GB/sec)[2]	% of measured peak[3]	execution time (msecs)	dominant utilization metric	resource band-width (GB/sec)[2]	% of measured peak[3]
fdtd_step1_kernel (FDTD-2D)	1	0.40	dram(9)	126.3	77.0%	0.57	dram(7)	88.4	74.3%	0.59	dram(8)	86.3	97.0%
	2	0.39	dram(9)	128.8	78.5%	0.47	dram(8)	108.0	90.8%	0.59	dram(8)	86.0	96.7%
	3	0.44	l2(10)	260.3	102.1%/66.1%	0.53	dram(8)	117.6	98.8%	0.59	dram(8)	86.1	96.7%
fdtd_step2_kernel (FDTD-2D)	1	0.47	dram(8)	109.8	66.9%	0.61	dram(6)	84.0	70.6%	0.59	dram(8)	86.4	97.0%
	2	0.39	dram(9)	130.2	79.4%	0.48	dram(8)	106.1	89.2%	0.59	dram(8)	86.0	96.7%
	3	0.47	l2(10)	283.0	111.0%/71.8%	0.59	dram(8)	119.6	100.5%	0.66	dram(7)	77.0	86.5%
fdtd_step3_kernel (FDTD-2D)	1	0.55	dram(8)	121.5	74.1%	0.73	dram(7)	92.7	77.9%	0.78	dram(8)	86.7	97.5%
	2	0.52	dram(8)	129.8	79.2%	0.66	dram(8)	102.5	86.2%	0.78	dram(8)	86.2	96.9%
	3	0.51	l2(10)	296.1	116.1%/75.1%	0.61	dram(9)	119.4	100.3%	0.78	dram(8)	86.6	97.3%
gemm_kernel (GEMM)	1	3.90	dram(8)	133.2	81.2%	6.93	dram(6)	81.3	68.4%	3.06	l2(10)	395.2	123.9%/77.3%
	2	6.46	tex(10)	332.2	98.9%	5.29	dram(8)	95.6	80.3%	5.66	tex_fu(8)	474.7	66.1%
	3	2.79	l2(10)	353.9	138.8%/89.8%	3.78	l2(10)	355.3	199.6%/116.1%	3.93	l2(10)	368.5	115.5%/72.1%
mm3_kernel1 (3MM)	1	3.79	dram(8)	138.1	84.2%	5.80	dram(7)	95.3	80.1%	3.06	l2(10)	395.1	123.8%/77.3%
	2	6.46	tex(10)	332.4	98.9%	5.62	dram(7)	90.4	76.0%	6.24	tex_fu(8)	430.5	60.0%
	3	2.76	l2(10)	328.0	128.6%/83.3%	3.74	dram(9)	112.4	94.4%	3.89	l2(10)	364.1	114.1%/71.2%
mm3_kernel2 (3MM)	1	3.79	dram(8)	136.2	83.0%	5.78	dram(7)	95.5	80.3%	3.06	l2(10)	395.9	124.1%/77.5%
	2	6.46	tex(10)	334.9	99.7%	5.62	dram(7)	87.4	73.4%	6.23	tex_fu(8)	430.6	60.0%
	3	2.77	l2(10)	316.0	123.9%/80.2%	3.73	l2(10)	373.6	209.9%/122.1%	3.90	l2(10)	363.0	113.8%/71.0%
mm3_kernel3 (3MM)	1	3.81	dram(8)	137.0	83.5%	5.79	dram(7)	95.3	80.1%	3.05	l2(10)	398.4	124.9%/78.0%
	2	6.46	tex(10)	334.9	99.7%	5.63	dram(7)	87.4	73.5%	6.23	tex_fu(8)	430.6	60.0%
	3	2.75	l2(10)	320.9	125.8%/81.4%	3.73	l2(10)	374.5	210.4%/122.4%	3.89	l2(10)	365.8	114.7%/71.6%
syr2k_kernel (SYR2K)	1	875.55	l2(10)	226.4	88.8%/57.5%	1367.23	ldst_fu(6)	N/A	-	227.66	l2(10)	330.7	103.7%/64.7%
	2	478.74	dram(4)	94.7	57.7%	962.07	dram(3)	42.4	35.7%	217.29	l2(8)	289.2	90.6%/56.6%
	3	204.44	l2(4)	86.7	34.0%/22.0%	233.69	dram(4)	45.6	38.4%	115.79	l2(7)	156.4	49.0%/30.6%
syrk_kernel (SYRK)	1	290.81	l2(10)	285.0	111.8%/72.3%	414.54	ldst_fu(5)	N/A	-	114.91	l2(9)	309.0	96.9%/60.5%
	2	221.21	l2(6)	160.2	62.8%/40.7%	139.75	l2(6)	217.6	122.3%/71.1%	82.77	l2(9)	302.9	95.0%/59.3%
	3	59.93	l2(6)	148.0	58.0%/37.6%	29.33	l2(7)	248.7	139.7%/81.3%	28.57	l2(9)	268.2	84.1%/52.5%

[1] implementation variant (1: unmodified, 2: scalar element texture, 3:four element vector texture)
[2] bandwidth on the most utilized resource as indicated by the utilization metrics
[3] ratio of attained bandwidth on the most utilized resource to the peak measured bandwidth (for L2, 1st value corresponds to load peak and 2nd to copy peak measurement)

recommended to study the exact memory access characteristics and patterns in order to decide which memory type is optimum for a particular usage. An issue that is left for future work is the performance implications on the potential combination of multiple fast memory types. In the overall, the micro-benchmark suite described in this work exposes useful insight to the fast memory types of the GPU as it yields peak measures of the effective bandwidth of each memory type, by using different element sizes.

ACKNOWLEDGMENT

This research was partially funded by the University of Athens Special Account of Research Grants no 10812.

REFERENCES

[1] AMD, *White Paper—AMD Graphics Cores Next (GCN) architecture*. AMD, June, 2012.

[2] Y. Cotronis, E. Konstantinidis, M.A. Louka and N.M. Missirlis, *A comparison of CPU and GPU implementations for solving the Convection Diffusion equation using the local Modified SOR method*. Parallel Computing, ISSN 0167-8191, Vol 40, No 7, pp. 173-185, 2014.

[3] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula and J. Cavazos, *Auto-tuning a high-level language targeted to GPU codes*, in Innovative Parallel Computing (InPar), 2012, pp. 1-10, 13-14 May 2012.

[4] Y. Huangfu and W. Zhang, *Profiling-based L1 Data Cache Bypassing to Improve GPU Performance and Energy Efficiency*, SIGBED Rev., issn 1551-3688, ACM, Vol. 12, No. 1, 2015, pp. 7-11, February 2015.

[5] W. Jia, K.A. Shaw, and M. Martonosi, *Characterizing and Improving the Use of Demand-fetched Caches in GPUs*, Proceedings of the 26th ACM International Conference on Supercomputing, ICS '12, isbn 978-1-4503-1316-2, San Servolo Island, Venice, pp. 15-24, 2012.

[6] E. Konstantinidis and Y. Cotronis, *Graphics processing unit acceleration of the red/black SOR method*. Concurrency and Computation: Practice and Experience, Vol. 25, No. 8, pp. 1107-1120, 2013.

[7] E. Konstantinidis and Y. Cotronis, *A Practical Performance Model for Compute and Memory Bound GPU Kernels*, Parallel, Distributed and Network-Based Processing (PDP), 2015 23rd Euromicro International Conference on, pp. 651-658, 2015.

[8] A. Li, G. van den Braak, A. Kumar, and H. Corporaal, *Adaptive and Transparent Cache Bypassing for GPUs*, Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '15, Austin, Texas, No. 17, pp. 17:1-17:12, 2015.

[9] X. Mei, K. Zhao, C. Liu and X. Chu, *Benchmarking the Memory Hierarchy of Modern GPUs*, 11th IFIP WG 10.3 International Conference, NPC 2014, Ilan, Vol. 8707, Taiwan, pp. 144-156, 2014.

[10] R. Meltzer, C. Zeng, and C. Cecka, *Micro-benchmarking the C2070*, GPU Technology Conference, 2013.

[11] NVidia, *Tuning CUDA Applications for Maxwell*. DA-07173-001_v7.0, NVidia, March 2015.

[12] NVidia, *NVidia CUDA C Programming Guide v. 7.0 Design Guide*. PG-02829-001_v7.0, NVidia, March 2015.

[13] NVidia, *Parallel Thread Execution ISA v4.2, Application Guide*. NVidia, March 2015.

[14] NVidia, *Profiler user's guide*. DU-05982-001_v7.0, NVidia, March 2015.

[15] Y. Tian, S. Puthoor, J.L. Greathouse, B.M. Beckmann, and D.A. Jiménez, *Adaptive GPU Cache Bypassing*, Proceedings of the 8th Workshop on General Purpose Processing Using GPUs, GPGPU-8, isbn 978-1-4503-3407-5, San Francisco, pp. 25-35, 2015.

[16] V. Volkov and J.W. Demmel, *Benchmarking GPUs to tune dense linear algebra*. In Proceedings of the 2008 ACM/IEEE conference on Supercomputing, SC'08, IEEE Press, Piscataway, NJ, USA, Article 31, pp. 1-11, 2008.

[17] D. Wodniok, A. Schulz, S. Widmer and M. Goesele, *Analysis of Cache Behavior and Performance of Different BVH Memory Layouts for Tracing Incoherent Rays*, 13th Eurographics Symposium on Parallel Graphics and Visualization, EGPGV '13, isbn 978-3-905674-45-3, Girona, Spain, pp. 57-64, Eurographics Association, 2013.

[18] H. Wong, M.M. Papadopoulou, M. Sadooghi-Alvandi and A. Moshovos, *Demystifying GPU microarchitecture through microbenchmarking*, Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on, IEEE, pp. 235-246, 2010.

[19] C. Yan, H. Yu, W. Xu, Y. Zhang, B. Chen, Z. Tian, Y. Wang, and J. Yin, *Memory bandwidth optimization of SpMV on GPGPUs*, Frontiers of Computer Science, Higher Education Press, issn 2095-2228, Vol. 9, No. 3, pp. 431-441, 2015.