Tomas Karnagel, Tal Ben-Nun, Matthias Werner, Dirk Habich, Wolfgang Lehner

Big data causing big (TLB) problems: taming random memory accesses on the GPU

Erstveröffentlichung in / First published in:

SIGMOD/PODS'17: International Conference on Management of Data. Chicago 14.-19.05.2017. ACM Digital Library, Art. Nr. 6.

DOI: https://doi.org/10.1145/3076113.3076115

Final edited form was published in "SIGMOD/PODS'17: International Conference on Management of Data. Chicago 2017", Art. Nr. 6 , ISBN 978-1-4503-5025-9
https://doi.org/10.1145/3076113.3076115

Big Data causing Big (TLB) Problems: Taming Random Memory Accesses on the GPU

Tomas Karnagel[⋆], Tal Ben-Nun[#], Matthias Werner[†], Dirk Habich[⋆], and Wolfgang Lehner[⋆]

[⋆]Database Systems Group
Technische Universität Dresden,
Germany
{first.last}@tu-dresden.de

[#]Department of Computer Science
Hebrew University of Jerusalem,
Israel
talbn@cs.huji.ac.il

[†]Information Services and HPC
Technische Universität Dresden,
Germany
matthias.werner1@tu-dresden.de

ABSTRACT

GPUs are increasingly adopted for large-scale database processing, where data accesses represent the major part of the computation. If the data accesses are irregular, like hash table accesses or random sampling, the GPU performance can suffer. Especially when scaling such accesses beyond 2GB of data, a performance decrease of an order of magnitude is encountered. This paper analyzes the source of the slowdown through extensive micro-benchmarking, attributing the root cause to the Translation Lookaside Buffer (TLB). Using the micro-benchmarks, the TLB hierarchy and structure are fully analyzed on two different GPU architectures, identifying never-before-published TLB sizes that can be used for efficient large-scale application tuning. Based on the gained knowledge, we propose a TLB-conscious approach to mitigate the slowdown for algorithms with irregular memory access. The proposed approach is applied to two fundamental database operations — random sampling and hash-based grouping — showing that the slowdown can be dramatically reduced, and resulting in a performance increase of up to 13x.

KEYWORDS

GPU, TLB, Random Memory Access, Virtual Memory, Grouping

ACM Reference format:
Karnagel, Ben-Nun, Werner, Habich, Lehner. 2017. Big Data causing Big (TLB) Problems: Taming Random Memory Accesses on the GPU. In *Proceedings of DaMoN'17, Chicago, IL, USA, May 15, 2017,* 10 pages.
DOI: http://dx.doi.org/10.1145/3076113.3076115

1 INTRODUCTION

Graphics Processing Units (GPUs) are increasingly used within heterogeneous hardware systems for large-scale query processing in databases [2, 5, 12]. Specifically, their hardware parallelism and memory access bandwidths contribute to considerable speedups. While most query operations show regular memory access patterns, like table or column scans, some operations exhibit highly irregular data access patterns. For example, a hash-based implementation of the group-by operator on the GPU utilizes data-dependent memory writes to fill a hash table [6]. If such operations are developed without considering all details of the GPU architecture, substantial slowdowns may occur. Currently, the most significant drop in performance for this kind of operator can be noticed when accessing GPU data larger than 2GB, which, in some cases, may result in a ≈13.3x runtime decrease (Section 2.1). This phenomenon has been previously observed in other works [4, 6], but up to now, only speculations were made about its origins.

To tackle this performance drop systematically, we examine two different query operations with irregular access patterns in this paper: *random sampling* and *hash-based grouping*. Both operations show a significant slowdown at 2GB. Based on an in-depth analysis with respect to the GPU memory hierarchy, we determine that the performance drop relates to the virtual memory system. Specifically, we identified the Translation Lookaside Buffer (TLB) as the source of this slowdown, where TLB misses cost hundreds of cycles per memory access. To understand the problem in detail, we formulate an elaborate micro-benchmarking methodology that uncovers the entire TLB hierarchy of a given GPU. The micro-benchmarks determine page size, number of TLB entries, number of hierarchy levels, TLB sharing across GPU multiprocessors, and the miss delay of each layer.

Using this knowledge, we propose *TLB-conscious data access*: a generalized, multi-stage approach for random memory accesses exceeding the TLB limits. We apply our solution to the two operations and show a performance benefit. Generally, we consider two different GPU architectures in this paper: (1) NVIDIA Kepler [15] and (2) NVIDIA Pascal [16]. With our benchmarks, we determine that the TLB hierarchy may vary from device to device. Consequently, we present how this variation is translated to differences in performance with respect to data size, and how the operations can be tuned to the underlying TLB hierarchy for increased efficiency.

Our contributions are:

(1) We identify TLB misses to cause potential slowdowns of up to 13.3x for GPU operations with random memory accesses.
(2) We propose a micro-benchmarking methodology to effectively determine the entire TLB hierarchy for a given GPU.
(3) We present findings of an unconventional TLB hierarchy, including a 3-level TLB and apparently different page sizes.
(4) We propose an easy-to-implement approach to mitigate the slowdown and show the effectiveness in our evaluation, yielding a performance increase of up to 13x.

Final edited form was published in "SIGMOD/PODS'17: International Conference on Management of Data. Chicago 2017", Art. Nr. 6 , ISBN 978-1-4503-5025-9
https://doi.org/10.1145/3076113.3076115

2 DATABASE GPU-OPERATIONS

In this section, we introduce our two investigated database operations, using large amounts of memory with an irregular memory access pattern. The two operations are *sampling of random values* and *hash-based grouping* as used in a group-by operator. In particular, we present and discuss the initial performance and scaling behavior with varying data sizes on two different GPU architectures. Table 1 summarizes the characteristics of the used GPUs.

2.1 Random Sampling

For random sampling, data is accessed at random positions to compute estimations like selectivities or aggregations (min, max, avg), without reading the full data set. We implemented this behavior by using a linear congruential generator (LCG) [7] to determine random data positions. LCGs are light-weight random number generators and do not add a significant computational overhead. Each GPU thread performs multiple read-only operations and adds the values to a sum, resulting in the following execution pattern:

```
for (i = 0; i < 1024; i++)
    sum += data[ next_Random_Position() ];
```

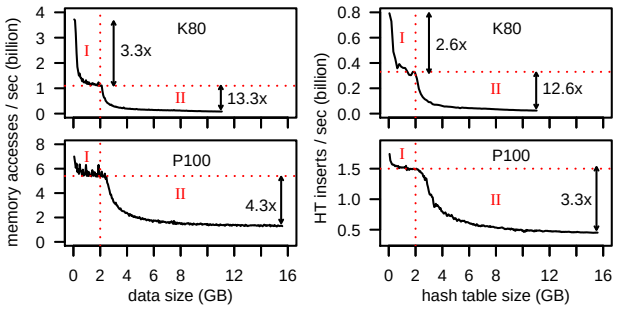
Each thread reads 1024 values at random positions and produces exactly one value as output. The kernel (GPU function) is started with enough threads to fully utilize the computational power of the given GPUs. In our evaluation, we kept the workload fixed, i.e., number of threads and memory accesses, but scale the memory region in which the data accesses are performed. The results for our two GPUs are depicted in Figure 1a. As we scale the memory region, we see two significant decreases in performance for the Kepler-K80 (region I and II) and one significant decrease for the Pascal-P100 (region II). Interestingly, the major slowdown starts at 2GB of memory for both GPUs (region II).

2.2 Hash-Based Grouping

Our grouping operation is based on a hash table, in which input data is aggregated. The input data can be accessed in a coalesced and GPU-friendly way, while the hash table accesses are pseudo-random. To evaluate possible effects, we implement a grouping prototype based on earlier work [6]. In our implementation, a large column of input data is divided into 128MB chunks, which are stored in CPU pinned main memory and accessed from the GPU using zero-copy (*Unified Virtual Addressing* [14]). We use the `Murmur3` hash-function [1] together with a fill factor of 0.5, assuming that the number of groups is known or estimated correctly. For hash table inserts, we use linear probing to access subsequent hash buckets until we match the current search key or an empty bucket. As

Name	Arch.	#Cores	Mem. size	Mem. bandwidth	Clock freq.
K80	Kepler	2x2496	2x12GB	2x 240GB/s	875 MHz
P100	Pascal	3584	16GB	732GB/s	1480 MHz

Table 1: GPU properties. The K80 has two GPUs on one board, however, we only use a single GPU for our tests to avoid interferences of both GPUs.



(a) Random Sampling **(b) Hash-Based Grouping**

Figure 1: Multiple slowdowns for random data accesses.

input data, we only use one column of integer values to minimize effects through data transfers and we build the hash table with two entries per hash-bucket (value, count) as it would be needed for the following query:

```
SELECT value, count(*) FROM values GROUP BY value;
```

In our evaluation, the input column has a size of 6GB (≈1.6 billion values), whereas the values are randomly distributed in a range of [0, #group). The number of distinct values (#group) is changing in the experiments, therefore, we change the size of the accessed hash table, while keeping the workload constant (always ≈1.6B hash tables accesses in total). The results are shown in Figure 1b. The performance is similar to the random sampling scenario, with two decreases in performance for the K80 (region I and II) and one major decrease for the P100 (region II). The performance is overall lower than random sampling, because one hash table insert includes reading the input value, applying the hash function, and multiple memory accesses in the hash table through linear probing to find the right hash bucket.

2.3 Result Discussion

To summarize, we observe similar results for random sampling and hash-based grouping, i.e., a significant performance loss for >2GB of memory (region II) for both GPUs and a smaller performance decrease on the K80 in region I. For both GPUs, memory accesses beyond 2GB are highly feasible, as 2GB are only $\frac{1}{6}$ (K80, 12GB per GPU) or $\frac{1}{8}$ (P100, 16GB) of the total available GPU memory.

As the slowdown is similar for both scenarios, we can exclude possible problem sources like input data access, the use of atomic functions, or overhead through the random number generation, as these operations only occur in one of the two applications. As the problem persists for multiple distinct scenarios, we can also exclude isolated programming issues. Therefore, the problem must be caused by the GPU architecture in combination with random memory accesses, the only common operation for both scenarios.

In our tests, we see that the P100 shows a significantly better performance than the K80, as it has a newer hardware architecture. However, the slowdown for memory accesses >2GB is still significant with factors of 4.3x for random sampling and 3.3x for grouping. To avoid this slowdown, we first need to identify the source by examining the GPU hardware architecture in detail.

Final edited form was published in "SIGMOD/PODS'17: International Conference on Management of Data. Chicago 2017", Art. Nr. 6 , ISBN 978-1-4503-5025-9
https://doi.org/10.1145/3076113.3076115

3 TLB BENCHMARK METHODOLOGY

In the previous section, we have seen a major performance drop for operations with random memory accesses at around 2GB on two different GPU architectures. The main question is now: *What is causing this effect?* The L2 data cache can not be responsible for the problem as its size is 1.5 MB (K80) and 4 MB (P100), far below the 2 GB border [6]. Through extensive micro-benchmarking, we found the virtual address translation and the TLBs to be the source of our performance slowdown. To determine the necessary TLB characteristics – properties and sharing between multiprocessors (SM) of a GPU – we developed two low-level benchmarks[1].

3.1 Virtual Memory on GPUs

GPUs use virtual addresses for their device memory for two reasons: (1) *Isolation*: The indirection controls a program's memory accesses and, thus, keeps it from disallowed memory accesses to internal device data or to data of other applications using the same GPU. (2) *Fragmentation*: Memory fragmentation can be hidden with virtual pages, allowing a large consecutive region of virtual memory to be scattered across many positions in physical memory. This can also increase memory bandwidth if physical memory is scattered to multiple memory chips, which then can be accessed in parallel. The translation of virtual to physical addresses is usually performed in pages, which are fixed blocks of memory. To access data within one page, a virtual page address is translated to a physical page address using a page table, while an additional offset defines the requested position inside the page. Address translation using the page table is costly. To reduce the page translation delays, TLBs cache virtual to physical address mappings. Generally, TLBs can be implemented for different page sizes and different amounts of entries. Unfortunately, NVIDIA does not publish any information about TLB sizes of their GPUs.

3.2 TLB property benchmark

To identify the TLB properties such as size or delays, we defined a single-threaded GPU kernel traversing a continuous data array in a specific *stride* for a specific *distance* (traversed data size), while performing data dependent accesses (pointer chasing). The kernel is always executed twice. The first run initializes the memory by loading data into the TLB, while the second run measures the execution cycles for memory accesses with initialized TLBs. We will measure low cycle counts if the data fits in the TLB (always TLB hit), while otherwise measuring high cycle counts due to TLB misses. To eliminate the influence of data cache misses, the accessed data for our kernel is stored in the L2 data cache, which also works with physical addresses. So even with the data being cached, the addresses have to be translated. Thus, any increase in our measurements is purely due to TLB misses.

We use our kernel to benchmark a given GPU with multiple strides and multiple distances, while searching for the pattern as shown in Figure 2. The depicted pattern consists of three stride sizes (X, $\frac{1}{2}X$, and $2X$). If we find this pattern, we can conclude that $X = page\_size$, $\frac{1}{2}X$ accesses every page twice, and $2X$ only accesses every second page. Each one of these stride sizes shows a low cycle count for smaller data sizes and a higher cycle count for larger ones, hence, this is a TLB border. If the stride size is equal to the page size (X), every data access requires a new address translation. TLB misses are encountered when the TLB can not hold all pages of the first execution run (>a in Figure 2).

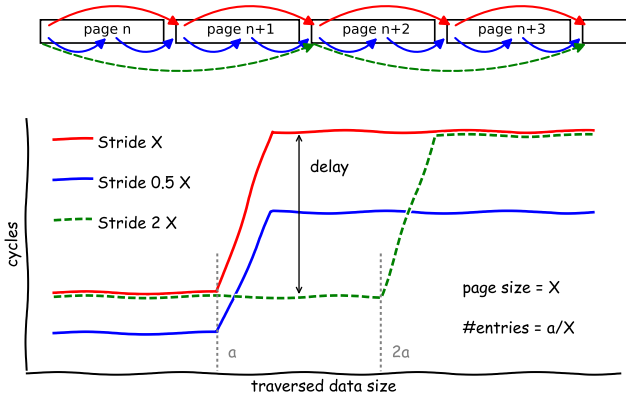


Figure 2: TLB Boundary pattern.

For a stride size of $\frac{1}{2}X$, every first access triggers a TLB miss, while the second access experiences a TLB hit, as it accesses the same page. A TLB hit is faster than a miss, so the average cycle count of all accesses is lower than X. However, it still accesses exactly the same pages as X (each page twice), which leads to TLB misses at exactly the same traversed data size (position a in Figure 2). Every stride size smaller than the page size behaves like $\frac{1}{2}X$: showing lower cycle counts but experiences the first TLB miss at the same position.

For a stride size of $2X$, every second page is accessed, leading to a TLB miss for double the traversed data size ($2a$). The low and high cycle counts are the same as X, because every access leads to an address translation and potentially to a TLB miss. Every stride size larger than the page size behaves like $2X$: showing similar cycle counts, while experiencing the first TLB miss at later positions.

3.3 TLB sharing benchmark

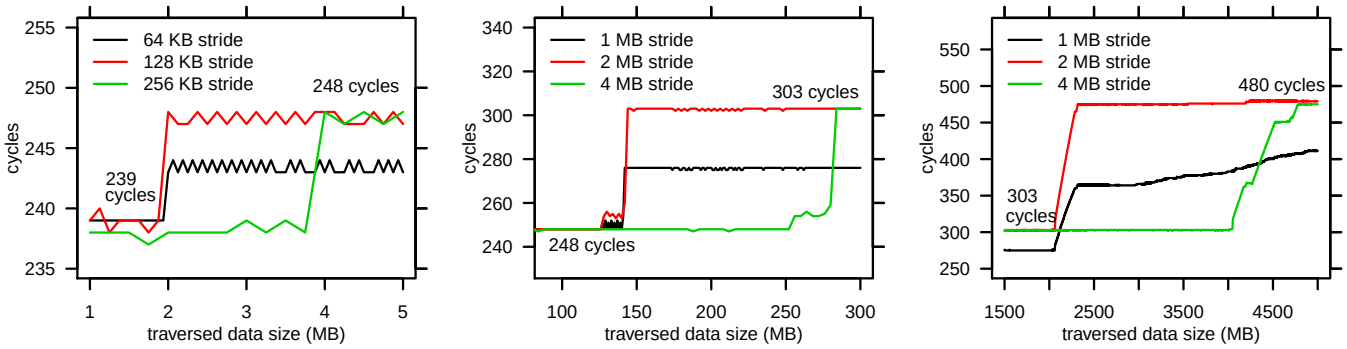
Our TLB sharing benchmark uses three stages: (1) accessing N pages on the i-th multiprocessor SM_i, (2) accessing N different pages on the k-th multiprocessor SM_k, and (3) accessing the first N pages again on SM_i. N is the number of pages that fit in the TLB, which should be tested. We measure the used cycles for the last stage. A low cycle count indicates no sharing between SM_i and SM_k, i.e., the accessed pages from the first stage are still in the TLB; whereas a high cycle count indicates TLB sharing, i.e., SM_k evicts the pages loaded by SM_i. To determine TLB sharing, we have to test every SM combination (#SM x #SM) for every TLB level.

4 BENCHMARK APPLICATION AND OBSERVATIONS

With our TLB benchmarks, we test different stride sizes until we find the pattern shown in Figure 2. In the following section, we present our findings separately for the TLB property benchmark and the TLB sharing benchmark.

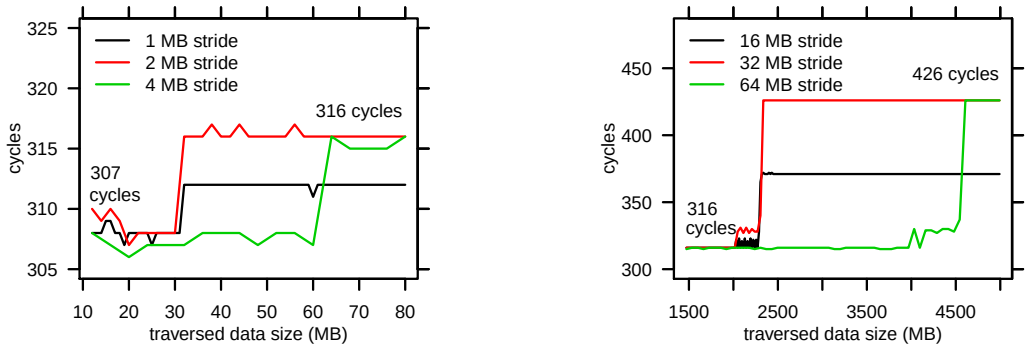
[1]Source code is available at: https://github.com/gcoe-dresden/cuda-gpu-tlb

Final edited form was published in "SIGMOD/PODS'17: International Conference on Management of Data. Chicago 2017", Art. Nr. 6 , ISBN 978-1-4503-5025-9
https://doi.org/10.1145/3076113.3076115



(a) L1 TLB boundary. (b) L2 TLB boundary. (c) L3 TLB boundary.

Figure 3: TLB boundary benchmark results for the K80.



(a) L1 TLB boundary. (b) L2 TLB boundary.

Figure 4: TLB boundary benchmark results for the P100.

4.1 TLB property benchmark results

K80: The results for the K80 are shown in Figure 3. We found our TLB boundary pattern three times for the K80, indicating three TLB levels. Figure 3a shows the L1 TLB with 2MB capacity and a page size of 128KB, indicating that the L1 TLB has 16 entries.

Figure 3b shows the L2 TLB, which is visible at 130MB as traversed data size. Interestingly, the benchmark identifies 2MB as page size, indicating that the L2 TLB has 65 entries. This means that the L2 TLB works with a 16x larger page size as the L1 TLB. This is also visible in the cycle counts. All stride sizes have the same cycle counts for data sizes <130MB. This indicates that all tested stride sizes access a new page for every data access (no two accesses to the same page). This is not the case for data >130MB, which hints to two different page sizes. The small noise-like area just around 130MB is probably caused by the TLB cache associativity. Due to this, some entries can be evicted before the TLB is full and some entries might stay in the TLB even when more memory is accessed. These associativity effects disappear with larger data, when clearly no entries can be kept in the TLB from the first iteration of our benchmark.

Figure 3c shows our search pattern for the L3 TLB, identifying 2MB as page size, 1032 TLB entries, and a traversed data size of 2064MB.

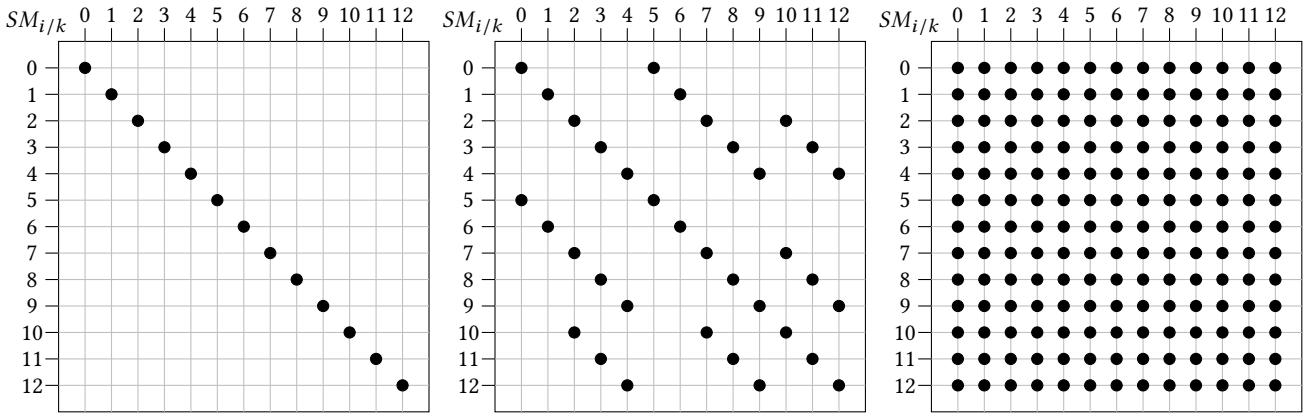
We can thus conclude that the K80 contains three TLB levels, with apparently different page sizes for the L1 TLB and the L2/L3 TLBs.

P100: The results for the P100 are shown in Figure 4. We found our TLB boundary pattern twice for the P100, indicating at least two TLB levels. For the L1 TLB (Figure 4a), the cycle count increases for 32MB traversed data sizes with a page size of 2MB. This results in 16 entries, which is the same amount of entries as for the L1 TLB of the K80. For the L2 TLB, we see the first increase in cycles at around 2080MB for a page size of 32MB, which is 16 times the size of the smaller page size. Similar to the K80, the P100 has 65 L2 TLB entries and the small noise-like behavior around the L2 TLB boundary (Figure 4b). Given the similarities to the K80, we assume that the L3 TLB boundary on the P100 is around 32GB (32MB page sizes and 1032 entries). Since our tested P100 GPU only contains 16 GB of memory, we can not test this TLB level.

4.2 TLB sharing benchmark results

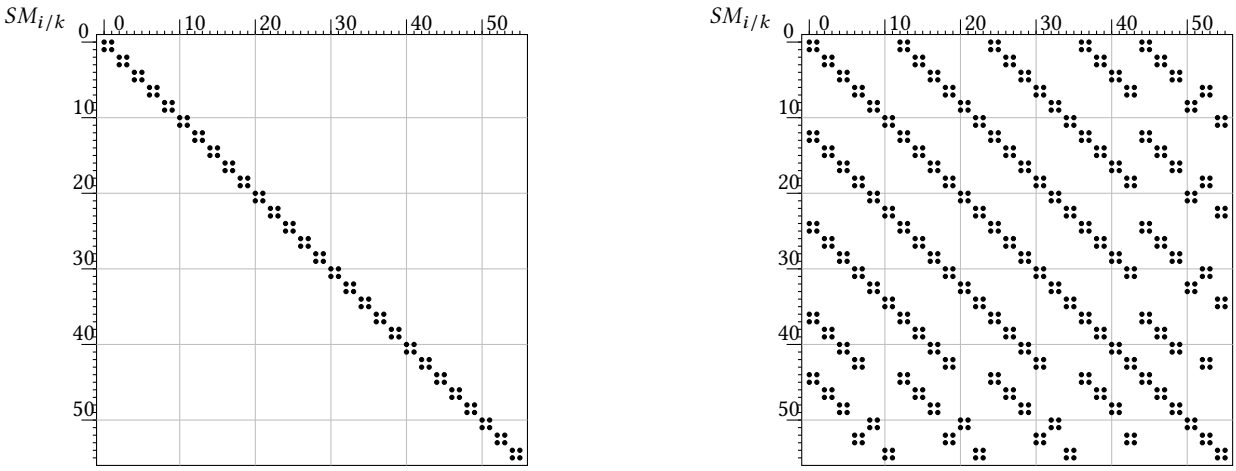
We also apply our TLB sharing benchmark using the gained knowledge of page sizes and TLB entries. We test every SM_i-SM_k combination and mark SM TLB sharing with ● .

Final edited form was published in "SIGMOD/PODS'17: International Conference on Management of Data. Chicago 2017", Art. Nr. 6 , ISBN 978-1-4503-5025-9
https://doi.org/10.1145/3076113.3076115



(a) L1 TLB sharing. **(b) L2 TLB sharing.** **(c) L3 TLB sharing.**

Figure 5: TLB sharing benchmark results for the K80 with 13 SMs (0-12).



(a) L1 TLB sharing. **(b) L2 TLB sharing.**

Figure 6: TLB sharing benchmark results for the P100 with 56 SMs (0-55).

K80: In Figure 5a, we see that only the SM itself can evict its L1 TLB entries, resulting in a diagonal pattern symbolizing a private L1 TLB.

For the L2 TLB, two or three SMs can interfere with each other, i.e., can evict each other's L2 TLB entries. We found that in general, the TLB is shared in a group of three SMs and for two groups, only two SMs share an L2 TLB. This can be explained by deactivated SMs on the GPU. The K80 GPU processor (*GK210*) is designed for 15 SMs, however, only 13 of them are activated. Based on the pattern from Figure 5b, we can add 2 hypothetical SMs between SM_9 and SM_{10} to reconstruct a regular pattern of three SMs sharing the L2 TLB as illustrated in Figure 7. Interestingly, the SMs that are deactivated can be different for every GPU, even if the GPU model is the same. We tested 256 different K80 GPUs and found 11 different configurations, caused by a different combination of deactivated SMs. The detailed results of these configurations are presented in Appendix A.

For the L3 TLB in Figure 5c, we see that every SM can evict L3 TLB entries of every other SM, therefore, we assume global L3 TLB sharing.

P100: For the P100, we can only test the L1 and L2 TLB sharing. Figure 6a shows the sharing behavior for the L1 TLB and Figure 6b depicts L2 TLB sharing. Instead of a private L1 TLB, each SM shares its L1 TLB with another SM. For the L2 TLB sharing, we observe that the TLB is usually shared between 10 SMs, with the exception of sometimes only 8 SMs sharing one L2 TLB. Again, this

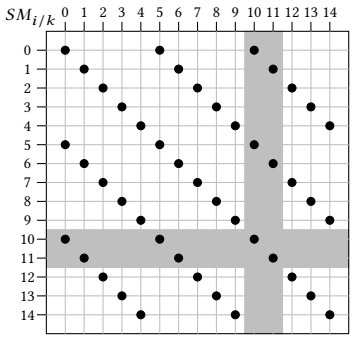


Figure 7: Hypothetical L2 TLB sharing with 15 SMs (hypothetical SMs are highlighted).

Final edited form was published in "SIGMOD/PODS'17: International Conference on Management of Data. Chicago 2017", Art. Nr. 6 , ISBN 978-1-4503-5025-9
https://doi.org/10.1145/3076113.3076115

can be explained with deactivated SMs. On the P100, only 56 SMs are activated, however, the architecture was designed for 60 SMs. Therefore, four SMs are deactivated, resulting in the non-uniform sharing of the L2 TLB. The reconstructed pattern of all 60 SMs is presented in Appendix B.

4.3 Observation Summary

The results of our benchmarks are summarized in Table 2 for both GPUs. The key observations are:

(1) We found three levels of TLBs for the K80 and two levels for the P100.
(2) For both GPUs, the different TLB levels apparently use different page sizes, where the L1 TLB uses a small page size and the L2/L3 TLB use a 16x larger page size.
(3) Compared to K80, the P100 always has 16x larger pages.
(4) For data larger than 2GB, the K80 has a total delay of 241 cycles, while the P100 only has a 119 cycle delay.

		L1 TLB	L2 TLB	L3 TLB
K80	Entries	16	65	≈1032
	Page Size	128 KB	2 MB	2 MB
	Cache-able Memory	2 MB	130 MB	≈2064 MB
	Delay on Miss	≈9 cycles	≈55 cycles	≈177 cycles
	TLB sharing	private	3 SM	global
P100	Entries	16	65	-
	Page Size	2 MB	32 MB	-
	Cache-able Memory	32 MB	2080 MB	-
	Delay on Miss	≈9 cycles	≈110 cycles	-
	TLB sharing	2 SM	10 SM	-

Table 2: TLB Findings for K80 and P100. "≈" indicates that the result is slightly varying or inconclusive.

4.4 Plausibility and Validation

As our results show unconventional TLB properties, we discuss their plausibility below. NVIDIA does not publish any information about TLB sizes on their GPUs. Thus, we have to rely on our results, but we found five points that strengthen our benchmark results:

First, the sizes of the L1 TLB (16 entries) and L2 TLB (65 entries) for Kepler GPUs (K80) were already reported by Mei et al. [10] and our measurements confirm their results. However, the authors did not report the different page sizes but only the page size of 2MB. This is understandable, because, even with 2MB strides, an L1 TLB miss occurs after 16 accesses. However, we found that smaller strides (down to 128KB) also need 16 accesses before a TLB miss occurs, indicating that a 2MB stride would only access every 16^{th} page.

Second, NVIDIA announced that the P100 GPUs work with 2MB page sizes [3]. We can confirm this for the L1 TLB, while the K80 already uses 2MB pages for the L2 and L3 TLB (as shown by [10]).

Third, the GPU cores are organized in a hierarchy, where one or more SMs are grouped to a *Texture Processing Cluster (TPC)* and multiple TPCs are combined to a *Graphics Processing Cluster (GPC)* [16]. The K80 has 1 SM per TPC and 3 SMs per GPC, while the P100 has 2 SMs per TPC and 10 SMs (5 TPC) per GPC [16]. For both GPUs, we see exactly the same hierarchy for our TLB sharing, indicating that every TPC has its own L1 TLB and every GPC has its own L2 TLB, while the L3 TLB is shared for all SMs.

Fourth, for both GPUs, we can see a significant performance drop in our investigated database operations when we access more data than ≈2GB. Even with different page sizes for both GPUs, we can pinpoint the problem to the L3 TLB on the K80 and the L2 TLB on the P100. We can even identify the L2 TLB boundary on the K80, where performance problems start at ≈130MB.

Fifth, in [6], the performance of a grouping operator on Kepler GPUs was improved by reducing the number of threads to <1000 instead of multiple thousands for data accesses beyond 2GB. With our results, we can explain that this is beneficial because each thread can load one page translation in the L3 TLB (1032 entries). The page translations stay in the TLB and since the threads use linear probing in the hash table, the translations can be reused. For more threads, page entries would be evicted during linear probing, resulting in multiple TLB misses per hash table access. We evaluated this approach on the K80 and the P100: (1) on the K80 with 1032 L3 TLB entries, we see the same effect, and (2) on the P100, no performance improvement is visible, as there <65 threads would be needed to avoid L2 TLB misses, however, these few threads severely underutilize the GPU and show a bad overall performance.

4.5 Arguments for Unconventional Properties:

There are mainly two unconventional results: (1) TLB entry numbers not being the power of two and (2) different page sizes for different TLB levels. The former was explained by Mei et al. for the L2 TLB [10]. 65 entries are the result of associativity optimizations, where six sets hold eight entries and one set holds 17 entries to store aligned page addresses. The L3 TLB could have similar optimizations resulting in the unconventional number of 1032 entries.

Different page sizes are already used in some CPU systems, where they can be stored in the same or different TLBs [11]. However, each page is allocated in one size or the other. For our results, different allocation sizes are not possible as all TLBs work with these allocations. We evaluated the allocation size and found that the smaller page size is always used for allocations (128KB on K80, 2MB on P100). One possible explanation for the apparently larger page sizes in the L2/L3 TLB could be a static pre-fetching algorithm, which always loads 16 contiguous pages when a TLB miss occurs. This would result in one TLB miss and 15 TLB hits, when using the small page size as traversal stride. Therefore, this optimization significantly reduces TLB misses as most applications with regular memory access scan large memory regions. For applications (and our benchmark), this can be interpreted as 16x larger page sizes, while in fact, only the reloading mechanism works on 16 pages at once. We have to emphasize that this is only our speculation. Without knowing the internal hardware configuration, we state the measured page sizes in Table 2.

5 TLB-CONSCIOUS DATA ACCESS

As a result of the previous section, we have to ensure a data access behavior according to the TLB properties for the given GPU to

Final edited form was published in "SIGMOD/PODS'17: International Conference on Management of Data. Chicago 2017", Art. Nr. 6 , ISBN 978-1-4503-5025-9
https://doi.org/10.1145/3076113.3076115

alleviate the observed slowdown for database operations with random memory accesses. To achieve such a data access, we propose an effective approach to reduce TLB misses called *TLB-conscious data access*. In this section, we introduce the general idea of our *TLB-conscious data access* and describe the application to our two database operations.

5.1 General Idea

Our overall approach is illustrated in Figure 8. For large memory regions, we channel the data accesses to be within a sub-section, which we call *TLB scope*. A TLB scope is smaller than or equal to the TLB capacity allowing an adjustment to different TLB properties. The data accesses of a GPU kernel are only executed when they fall within the current TLB scope, while other accesses are ignored. To access the whole memory region, the GPU kernel needs to be executed multiple times, whereas each pass uses a different TLB scope. This is beneficial for two reasons: (1) The large slowdown of random memory access beyond the TLB capacity is avoided; and (2) for each kernel pass, only a fraction of the read or write operations is executed, reducing the overall memory accesses per pass. In total, we have the same amount of memory accesses compared to the traditional approach, but there is overhead in form of redundant computation and additional kernel launches. However, this overhead should be low compared to TLB misses based on the provided computational power.

5.2 Application to Random Sampling

We apply our approach to random sampling by changing the kernel towards accessing a memory location only if it resides in a predefined TLB scope, as shown in the following execution pattern:

```
for (i = 0; i < 1024; i++){
    position = next_Random_Position();
    if ( in_current_TLB_Scope(scopeNumber, position) )
        sum += data[ position ];
}
```

Depending on the size of the data and the size of the TLB scope, the kernel needs to be executed multiple times. For example, 4GB of data can be divided into two 2GB TLB scopes, therefore, the kernel has to be executed two times, every time restricted to only one scope. The kernel evaluates the access position before any data access. When starting a kernel pass, the current TLB scope number needs to be specified. Furthermore, it is crucial to initialize the random number generator in exactly the same way, to compute the same random numbers for each execution and the result sum needs to be kept globally for all kernel executions.

5.3 Application to Hash-Based Grouping

For hash-based grouping, the application is similar, with the difference that input data also has to be accessed. We propose two different approaches to apply our *TLB-conscious data access*: (1) a naive approach, similar to random sampling, and (2) a copy-based approach with further optimizations.

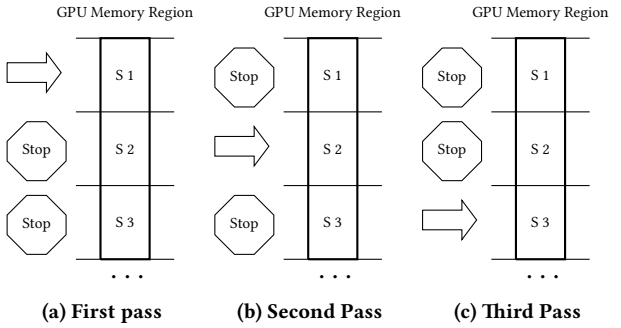


Figure 8: TLB-conscious data access with multiple passes to access the whole memory region.

Naive Approach: For the naive approach, the hash table is divided into smaller TLB scopes and the kernel is executed multiple times. This avoids L2/L3 TLB misses for the hash table, however, the input data is scanned multiple times, resulting in multiple zero-copy transfers over the PCI-Express (PCIe) bus.

Copy Approach: To avoid multiple zero-copy transfers, we can first copy the data to the GPU and then apply the naive approach. We follow this idea, but merge the copy operation with the first pass over the data for efficiency (Figure 9). The first pass accesses the data through zero-copy as before. If data can be written in the first TLB scope, we perform the hash table access. If it needs to be written to another TLB scope, we store it in a separate memory buffer on the GPU to be processed later. All following passes can use the data already stored on the GPU and avoid the memory transfer. Storing the input data on the GPU is only a small space overhead, as our grouping operator natively works on 128MB strides of input data at one time. This means that only 128MB of input data need to be stored on the GPU at any time, even if the original input column is much larger.

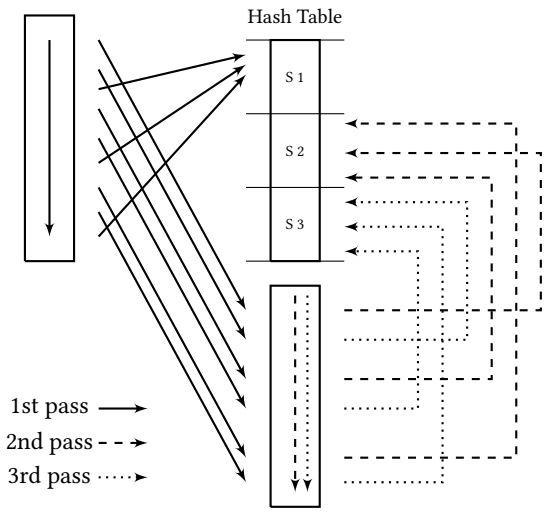
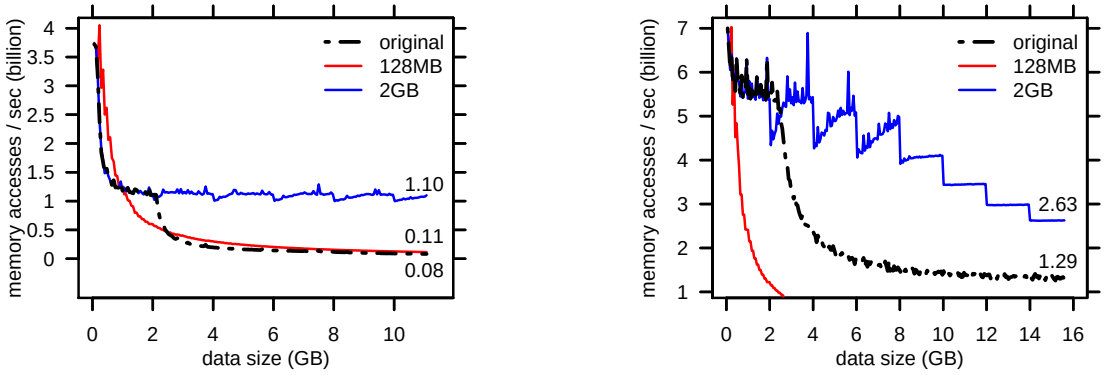


Figure 9: TLB-conscious grouping: copy approach.

Final edited form was published in "SIGMOD/PODS'17: International Conference on Management of Data. Chicago 2017", Art. Nr. 6 , ISBN 978-1-4503-5025-9
https://doi.org/10.1145/3076113.3076115



(a) K80 (b) P100

Figure 10: Random sampling with different TLB scope sizes.

6 EVALUATION

We evaluate our *TLB-conscious Data Access* for random sampling and the hash-based grouping, both on the K80 and the P100. We use two different scope sizes: 128MB, potentially avoiding L2 TLB misses on the K80, and 2GB, potentially avoiding L3 TLB misses on the K80 and L2 TLB misses on the P100.

It is worth noting that smaller TLB scopes were also tested, in order to avoid L1 TLB misses. However, the large amount of necessary passes, combined with the small delay of an L1 TLB miss, prevent performance improvement in those cases.

6.1 Random Sampling

Figure 10 illustrates the results for random sampling. The K80 results are shown in Figure 10a. For the small TLB scopes of 128MB, the performance is decreasing with increasing data sizes, as more and more passes are necessary. A minor improvement is seen for data sizes between 130MB and 1GB, as 128MB scopes avoid L2 TLB misses as well as L3 TLB misses. However, with more than 8 passes, the speedup disappears due to the additional pass overhead. For larger data sizes (>2.5GB), there is a small speedup again, as the original version suffers from L3 TLB misses. 2GB scopes only avoid L3 TLB misses, while the larger scope size results in fewer passes than for 128MB. For 2GB TLB scopes, the pass overhead is not significant due to the lower number of necessary passes, and the avoidance of L3 TLB misses outweighs the additional overhead. The final speedup is 13x.

The P100 results are shown in Figure 10b. 128MB TLB scopes are not beneficial due to their pass overhead and due to not avoiding L1 TLB misses. For the P100, the overhead of multiple passes is more significant, as the delay of an L2 TLB miss is smaller than L3 TLB misses on the K80. This is clearly visible for 2GB scopes. In this case, with every new pass, the performance first decreases, mostly due to memory accesses being in the first pass while adding the computational overhead of the second pass (e.g., around 2GB). When accessing a wider memory range, data accesses are equally divided between the passes, hiding the pass overhead and leading to better performance again (e.g., around 4GB). At some point (>8GB) the execution is purely bound by the pass overhead, leading to a performance decrease with every new pass and a constant performance for a constant amount of passes. Even in this case, a speedup of 2x can be achieved for accesses of 16GB.

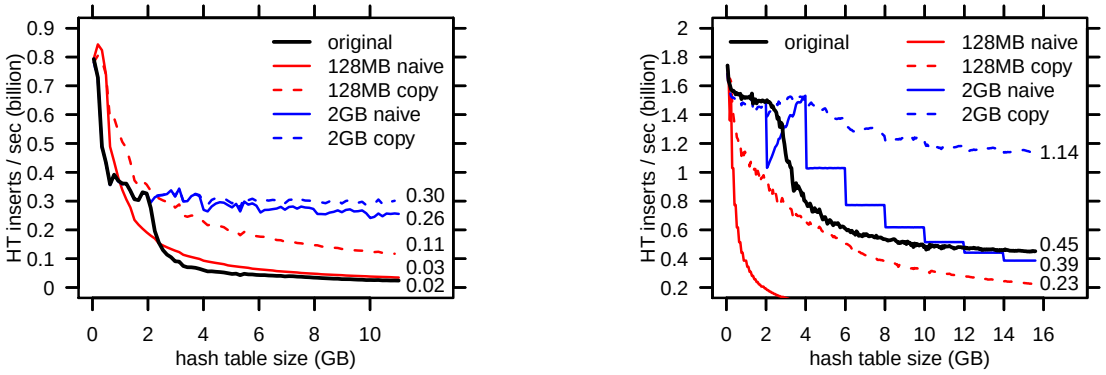
6.2 Hash-Based Grouping

For the grouping operator, we evaluate both TLB scope sizes and the two proposed approaches: *naive*, where data is accessed through the PCIe bus for every pass, and *copy*, where data is copied within the first pass, while subsequent passes do not use the PCIe bus again.

The K80 results are shown in Figure 11a. Again, 128MB scopes exhibit a minor speedup on the K80. We test hash tables sizes up to 11.2GB, where 128MB scopes result in 89 passes. Applying our naive approach, each pass transfers 6GB of data from the CPU to the GPU, which results in effectively transferring $89 * 6GB = 534GB$ through the PCIe. It is interesting that even this approach shows a small speedup compared to one pass transferring only 6GB and accessing the entire hash table. When optimizing this through our data copy approach, the performance increases significantly, as we can exchange up to 88 PCIe data transfers with global memory reads on the GPU. For 2GB TLB scopes, the copy approach is also better than the naive approach. However, the improvement is not that substantial, as we can only omit up to 5 PCIe transfers, which is insignificant compared to the original problem. The final speedup is 12.5x.

For the P100, the results are slightly different (Figure 11b), because the extent of the problem is not as vast as for the K80. Either version with 128MB scopes is slower than the original version, as multiple passes over the data are too time consuming, even when using our copy approach. We see a significant difference in the two approaches for 2GB scopes, as the execution is limited by PCIe transfers for three passes or more. Without the additional transfers (copy approach), we achieve a final speedup of 2.5x.

Final edited form was published in "SIGMOD/PODS'17: International Conference on Management of Data. Chicago 2017", Art. Nr. 6 , ISBN 978-1-4503-5025-9
https://doi.org/10.1145/3076113.3076115



(a) K80 **(b) P100**

Figure 11: Grouping results for different TLB scope sizes.

7 RELATED WORK

Caching physical addresses with TLBs has been studied for main memory database systems and has been found to be a performance critical factor on CPUs [8, 9]. For GPUs, the problem with random reads on more than 2GB was also reported for hash joins [4] and groupings [6]. In both works, the TLB was not identified as the source of the problem. Mei et al. [10] benchmark the GPU memory hierarchy for the NVIDIA Fermi, Kepler, and Maxwell architectures with some TLB-related tests for the L1 and L2 TLB, including associativity. Papadopoulou et al. [13] benchmark an older GT200 GPU, reporting an L1 TLB with 16 entries, an L2 TLB with 64 entries, and a third TLB structure they could not measure, all using 512 KB as page size. The authors state that the L2 TLB shows incomprehensible associativity behavior, which could indicate 65 entries instead of 64 [10]. As we showed, the page sizes vary between different architectures, so 512 KB for the larger pages on an older GPU is realistic.

8 CONCLUSION

In this paper, we evaluated two data-intensive operations with irregular data access patterns and show that the performance decreases for larger data sizes. More recent GPU architectures, such as the P100, mitigate this effect, however, the performance decrease is still significant. Through extensive micro-benchmarking, we found the source of the problem to be the TLB hierarchy, where data access beyond a TLB's ability to cache pages results in a TLB miss, slowing down the execution. We present our benchmarking methodology in detail and report the benchmark results for the K80 and P100 GPU. Interestingly, we found three levels of TLBs and two different page sizes per GPU. Based on the newly gained hardware knowledge, we propose TLB-conscious data access, which reduces TLB misses through redundant work. We apply this approach successfully to the two data-intensive operations and show speedups of up to 13x on the K80 and up to 2.5x on the P100.

ACKNOWLEDGMENTS

This work is funded by the German Research Foundation (DFG) within the Cluster of Excellence "Center for Advancing Electronics Dresden" (Orchestration Path), and supported by the DFG German Priority Programme 1648 "Software for exascale Computing" (SPPEXA), project FFMK. Parts of the evaluation hardware were generously provided by Dresden GPU Center of Excellence. We also thank the OS group of TU Dresden for the constructive discussions on TLB architectures.

REFERENCES

[1] Austin Appleby. 2008. Murmurhash project. (2008). http://code.google.com/p/smhasher/.
[2] Max Heimel, Michael Saecker, Holger Pirk, Stefan Manegold, and Volker Markl. 2013. Hardware-oblivious parallelism for in-memory column-stores. *Proceedings of the VLDB Endowment* 6, 9 (2013), 709–720.
[3] Jeremy Appleyard. 2016. *Nvidia Presentation: PASCAL AND CUDA 8.0.*
[4] Tim Kaldewey, Guy Lohman, Rene Mueller, and Peter Volk. 2012. GPU Join Processing Revisited. In *Proceedings of the Eighth International Workshop on Data Management on New Hardware (DaMoN '12)*. ACM, New York, NY, USA, 55–62.
[5] Tomas Karnagel, Dirk Habich, and Wolfgang Lehner. 2017. Adaptive Work Placement for Query Processing on Heterogeneous Computing Resources. *PVLDB, Vol. 10, No. 7* (2017).
[6] Tomas Karnagel, Rene Mueller, and Guy M. Lohman. 2015. Optimizing GPU-accelerated Group-By and Aggregation. In *ADMS at VLDB*.
[7] Donald E. Knuth. 1997. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
[8] Stefan Manegold. 2002. *Understanding, modeling, and improving main-memory database performance.*
[9] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. 2000. Optimizing Database Architecture for the New Bottleneck: Memory Access. *The VLDB Journal* 9, 3 (Dec. 2000), 231–246. DOI: http://dx.doi.org/10.1007/s007780000031
[10] Xinxin Mei and Xiaowen Chu. 2017. Dissecting GPU memory hierarchy through microbenchmarking. *IEEE Transactions on Parallel and Distributed Systems* 28, 1 (2017), 72–86.
[11] Sparsh Mittal. 2016. A survey of techniques for architecting TLBs. *Concurrency and Computation: Practice and Experience* (2016).
[12] Todd Mostak. 2013. An Overview of MapD (Massively Parallel Database). *White Paper* (2013).
[13] Misel myrto Papadopoulou, Maryam Sadooghi-alvandi, and Henry Wong. 2009. *Micro-benchmarking the GT200 GPU.* Technical Report.
[14] NVIDIA 2015. *CUDA C Programming Guide* (7.0 ed.). NVIDIA.
[15] NVIDIA. 2015. *TESLA K80 GPU Accelerator - Board Specification* (BD-07317-001_v05 ed.).
[16] NVIDIA. 2016. *NVIDIA Tesla P100 - The Most Advanced Datacenter Accelerator Ever Built Featuring Pascal GP100, the World's Fastest GPU* (WP-08019-001_v01.1 ed.).

Final edited form was published in "SIGMOD/PODS'17: International Conference on Management of Data. Chicago 2017", Art. Nr. 6 , ISBN 978-1-4503-5025-9
https://doi.org/10.1145/3076113.3076115

APPENDIX

A TLB SHARING FOR 256 K80 GPUS

To get a better understanding of the TLB sharing patterns, we evaluated our TLB sharing benchmark on 256 different yet theoretically identical K80 models. The L1 TLB and the L3 TLB patterns are fixed: L1 TLBs are always private for one SM and L3 TLBs are always shared between all SMs. However, the L2 TLB shows different patterns according to the deactivated SMs. In theory, there could be 105 different combinations for two deactivated SMs out of 15. The configuration statistics are shown in Table 3. The shown pattern in Figure 5b is by far the most common one, used in 161 of the 256 tested GPUs. Most GPUs have the deactivated SMs in two different sharing groups, leaving two groups of 2 SMs that share the L2 TLB (instead of 3 SMs). Additionally, we found one occurrence, where two SMs of one group where deactivated (SM 5 and 10 missing), leaving 1 SM to have its own L2 TLB, while the other L2 TLBs are shared each by three SMs. Depending on the workload, this might result in different performance of the SMs when either one, two, or three SMs share an L2 TLB.

B HYPOTHETICAL L2 TLB SHARING FOR THE P100

Figure 6b showed an incomplete pattern of L2 TLB sharing, caused by deactivated SMs. The four deactivated SMs are missing between SM 43 and 44. The complete pattern is shown in Figure 12. To achieve this complete pattern, the four missing SMs need to be added (in gray) and the TPC containing SM 50 and 51 need to be swapped with the TPC containing 53 and 54 (in orange). In our TLB sharing benchmark, we identify the SMs according to the *smid* register, because we can not proactively schedule a thread on a specific SM. Since these SM-IDs are logical, they are not directly dependent on the SM's location and IDs might be configured in different order for any reason. Unfortunately, we do not have multiple P100 GPUs, so we can not verify if these SMs are always swapped, and which SMs are usually deactivated.

Deactivated SMs	Occurrence	Percentage
10, 11	161	62.9%
10, 13	36	14.1%
10, 12	23	9.0%
10, 14	17	6.6%
12, 14	4	1.6%
11, 14	4	1.6%
11, 12	4	1.6%
13, 14	2	0.8%
11, 13	2	0.8%
12, 13	2	0.8%
5, 10	1	0.4%
total	256	≈100%

Table 3: Deactivated SMs (0-14) for 256 K80 GPUs.

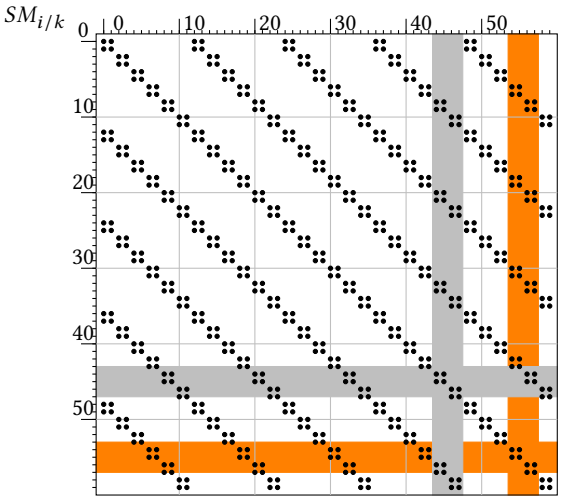


Figure 12: Hypothetical L2 TLB sharing with 60 SMs (0-59). Four SMs were added (gray) and four SMs were swapped (orange).