2018 IEEE International Parallel and Distributed Processing Symposium

WarpDrive: Massively Parallel Hashing on Multi-GPU Nodes

Daniel Jünger
Institute for Computer Science
Johannes Gutenberg University
Mainz, 55128 Germany
Email: djuenger@students.uni-mainz.de

Christian Hundt
Institute for Computer Science
Johannes Gutenberg University
Mainz, 55128 Germany
Email: hundt@uni-mainz.de

Bertil Schmidt
Institute for Computer Science
Johannes Gutenberg University
Mainz, 55128 Germany
Email: bertil.schmidt@uni-mainz.de

***Abstract*—Hash maps are among the most versatile data structures in computer science because of their compact data layout and expected constant time complexity for insertion and querying. However, associated memory access patterns during the probing phase are highly irregular resulting in strongly memory-bound implementations. Massively parallel accelerators such as CUDA-enabled GPUs may overcome this limitation by virtue of their fast video memory featuring almost one TB/s bandwidth in comparison to main memory modules of state-of-the-art CPUs with less than 100 GB/s. Unfortunately, the size of hash maps supported by existing single-GPU hashing implementations is restricted by the limited amount of available video RAM. Hence, hash map construction and querying that scales across multiple GPUs is urgently needed in order to support structured storage of bigger datasets at high speeds.**

In this paper, we introduce *WarpDrive* – a scalable, distributed single-node multi-GPU implementation for the construction and querying of billions of key-value pairs. We propose a novel subwarp-based probing scheme featuring coalesced memory access over consecutive memory regions in order to mitigate the high latency of irregular access patterns. Our implementation achieves 1.4 billion insertions per second in single-GPU mode for a load factor of 0.95 thereby outperforming the GPU-cuckoo implementation of the CUDPP library by a factor of 2.8 on a P100. Furthermore, we present transparent scaling to multiple GPUs within the same node with up to 4.3 billion operations per second for high load factors on four P100 GPUs connected by NVLink technology. WarpDrive is free software and can be downloaded at https://github.com/sleeepyjack/warpdrive.

I. Introduction

Hash maps are ubiquitous in informatics and almost every field of natural science. This includes the representation and processing of sparse data such as bag-of-words models [1], the efficient intersection of voxelized geometric objects [2], and approximate nearest neighbor computation [3]. Furthermore, hashing has manifold applications in bioinformatics, e.g. almost duplicate detection in metagenomic classification [4], [5], or seed index construction during short read mapping onto reference genomes [6].

With the ongoing prevalence of Big Data technologies, size and availability of recorded data are expected to grow rapidly in the foreseeing future. As a result, efficient hash map implementations are of high importance to a variety of applications.

Massively parallel accelerators have been widely adopted for number crunching due to their vast compute capability and highly competitive compute-to-energy ratio. Besides that, many application with regular memory access patterns benefit additionally from their high bandwidth video memory which outperforms common memory modules of traditional workstations by almost one order-of-magnitude. Unfortunately, those performance gains may vanish for memory-bound data-intensive algorithms with highly irregular access patterns. This is the case for hash-based index construction: even though insertion of large numbers of key-values pairs can be performed in parallel, associated main memory accesses are slow due to their random nature. Nevertheless, a number of approaches have been proposed for GPU-based hashing using sophisticated probing schemes and memory access techniques [2], [7]–[9]. However, speedups in comparison to state-of-the-art CPU implementations [10] are only modest. Furthermore, all these approaches are restricted to a single GPU. Thus, the supported in-core hash map size is bounded by the available global memory which limits the applicability for processing large-scale datasets.

The contributions of this paper are three-fold.

1) We introduce a novel GPU-based hash map algorithm featuring an unprecedented subwarp-level probing scheme tailored to suit the characteristics of high latency and high bandwidth video memory which outperforms existing single-GPU approaches by a factor of approximately 3 for high load factors $\alpha \geq 95\%$.
2) We propose a scalable multi-GPU hash map construction/querying scheme that allows for the building of hash maps containing several billions of key-value pairs by exploiting fast GPU interconnection networks within a single node. Using a compute node with four Tesla P100 GPUs connected by NVLink technology we are able to process 32 GB of data in less than 2 seconds.
3) We present an asynchronous technique that allows for the execution of out-of-core insertion and retrieval operations from/to the CPU by overlapping data transfers over PCIe/NVLINK and computation in order to reduce the impact of expensive communication primitives.

The rest of this paper is organized as follows. Background on hashing and associated parallelization strategies is provided in Section II. Section III discusses related work. Our single-

1530-2075/18/$31.00 ©2018 IEEE
DOI 10.1109/IPDPS.2018.00054

GPU and multi-GPU hash map construction algorithms are presented in Section IV. Section V discusses experimental results. Section VI concludes the paper.

II. Background

Hash maps allow for the modelling of exact functional dependencies $f : K \to V$, $k \mapsto f(k) := v$ mapping a sparse domain K onto its associated image space V. In contrast to dense look-up tables which reserve dedicated memory for every potential key $k \in K$, hash maps compress the sparse domain K into an index set I by means of a hash function $h : K \to I$, $k \mapsto h(k) := i$ which assigns a memory position i to each key k. While bijective mappings between K and I can be realized with *minimal perfect hash functions*, they often suffer from high computation time and the assumption that the complete set of keys K is known a priori. Alternatively, h can be chosen as unfaithful (non-injective) map which introduces potential collisions $h(k) = h(k')$ for two distinct keys $k, k' \in K$. Those ambiguities have to be resolved using an appropriate *collision resolution* technique. Among the most popular strategies are:

- **Chaining:** two or more ambiguous keys are stored in a bucket residing at the same position i. This involves either cache-inefficient pointer chasing in case of linked lists or memory over-subscription in case of fixed-length arrays.
- **Open Addressing:** stores colliding entries in distinct locations by means of a deterministic probing scheme which traverses a sequence of positions in a dedicated order.

From a parallelization point of view, open addressing is usually preferable over chaining since updates on key-value pairs (k, v) can often be accomplished efficiently in an atomic manner while lock-free lists waste valuable memory by storing a pointer for each node. Race condition-free insertion and deletion of nodes in linked lists is error-prone due to (in)famous pitfalls of lock-free programming such as the ABA problem and priority inversion [11]. Furthermore, open addressing hash maps can be extended to multi-value hash maps in a straightforward manner. Hence, we focus on open addressing throughout the rest of this paper.

While x86_64 CPUs support *compare-and-swap* (CAS) instructions for up to 128 consecutively stored bits, massively parallel accelerators such as CUDA-enabled devices are limited to 64-bit words. As a result, we have to pack key-value pairs (k, v) into 64 bits using an *array of struct* (AOS) memory layout. If all bits are needed for the key one can alternatively store the arrays of keys K and values V separately as *struct of arrays* (SOA). The latter variant uses relaxed reads and writes to the value array which might introduce priority inversion in case of simultaneously inserting distinct values for the same key. Another advantage of AOS is its cache-friendly access pattern during the querying phase (see Fig. 1).

Open addressing hash maps are equipped with a deterministic probing scheme which specifies a sequence of to be probed slots in case the initial position $h(k)$ is already occupied by a colliding entry $h(k') = h(k)$. The majority of

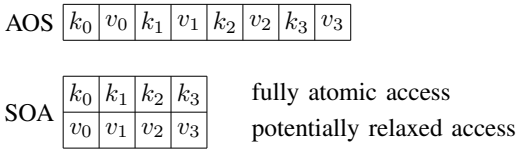

Fig. 1. Memory layouts for open addressing hash maps. AOS ensures cache-friendly and fully atomic access onto key-value pairs up to 64 bits. In contrast, the separated key and value arrays in the SOA format allow for longer keys at the cost of inferior caching and potential priority inversion during updates.

implementations employ one or a combination of the following probing strategies. Let l be the number of probing attempts, c be the capacity of the hash map, and $s(k, l)$ the l-th element in the sequence where $s(k, 0) = h(k)$ then:

- **Linear Probing:** searches an unoccupied slot (either empty or deleted) in the direct neighborhood of $h(k)$. This results in the probing sequence $\big(s(k,l)\big)_{l \in \mathbb{N}_0}$ where:
$$s(k, l) = \big(h(k) + l\big) \bmod c \quad . \tag{1}$$
- **Quadratic Probing:** aims to escape crowded regions typically caused by linear probing using quadratic steps:
$$s(k, l) = \big(h(k) + l^2\big) \bmod c \quad . \tag{2}$$
- **Chaotic Probing/Double Hashing:** uses a completely random but reproducible step pattern. Let $g(k)$ be a second hash function then:
$$s(k, l) = \big(h(k) + l \cdot g(k)\big) \bmod c \quad . \tag{3}$$

While linear probing is cache-efficient, it tends to produce unreasonably long sequences in occupied regions. Quadratic and chaotic probing avoid so-called *primary clustering* using larger step sizes at the cost of more cache misses. From a theoretical point of view, it can be shown that in general linear probing provides only weak guarantees: a pair-wise independent hash function ensures expected logarithmic time for hash table operations whereas a 5-wise independent hash function warrants operations in constant time [12]. The latter can be constructed using tabulation based hashing schemes [13]. Nevertheless, double hashing with two pair-wise independent hash functions ensures comparably strong properties [14]. Note that straightforward extensions of the aforementioned probing schemes have been proposed in the literature: among others Cuckoo Hashing [2], Robin Hood Hashing [15], and Hopscotch Hashing [16].

Fully featured (sequential) hash map implementations such as `std::unordered_map` from the C++11 standard library support on-demand resizing in case the number of inserted elements n exceed the capacity c. A common strategy is to reinsert the whole data structure into a new instance if the *load factor* $\alpha = \frac{n}{c}$ reaches a critical value of e.g. 90%. Moreover, keys and values of arbitrary size can be inserted and deleted at any time. The situation is more complex in a parallel context: modifications of the same slot have to be serialized using either slow global mutexes or efficient CAS operations on key-value pairs of limited length. This paper focuses on the latter. While

insertions and queries can in general be issued concurrently without violating the integrity of the hash map, the actual outcome is determined by the current event horizon of the data structure, i.e. a to be inserted key that is queried at the same time might be returned or not depending on which operation wins the race. Our implementation features robust execution times even for high load factors $\alpha > 95\%$. In the unlikely case that the probing scheme cannot determine an empty slot for $n < c$ the whole data structure is invalidated followed by a subsequent reconstruction with a distinct hash function.

A competing (concurrent) data structure to open addressing hash maps are key-value stores based on sort-and-compress approaches. The keys are sorted together with their associated values using an efficient sorting algorithm such as CUDA Unbound's radix sort primitive [17]. Multiple values belonging to the same key (in case of multi-value hash maps) are subsequently compressed using a logarithmic time parallel prefix scan. Querying can be accomplished in logarithmic time with a binary search. A major drawback of sort-and-compress based techniques is their memory consumption: both sorting and prefix computation usually require $\mathcal{O}(n)$ auxiliary memory which effectively reduces the capacity by a factor of two. Open addressing hash maps require memory for only $c \approx n$ slots while operating at the same or superior speed.

III. Related Work

Hash map related research dates back to the very beginning of computer science. Early research elaborates on the theoretical foundations of hashing and the efficient construction of sequential hash maps. The multitude of publications cannot be covered within the scope of this manuscript. For the sake of brevity, Don Knuth's *notes on open addressing* [18] shall serve as representative for this era.

The focus of hash map research has been shifting with the introduction of novel hardware architectures and data acquisition technologies. The emergence of affordable multi-core CPUs and programmable GPUs in the consumer market has been perceived as a major game changer. In the early 90s, Matias et al. [19] laid the theoretical foundations of concurrent hashing on Parallel Random Access Machines. Maier et al. [10] recently proposed *Folklore* – a scalable concurrent hash map suite for multi-core CPUs. Their implementation employs CAS operations on fixed-length machine words and achieves a performance of up to 300 million insertions per second on a 24-core dual-socket workstation with 48 threads. On that task, *Folklore* outperforms competing CPU implementation including Intel's `tbb::concurrent_hash_map` from the Threading Building Blocks (TBB) library. Other notable but less scalable multi-core implementations include *Junction* [20], and *libcuckoo* [21].

In comparison, hash map implementations on massively parallel architectures such as CUDA-enabled accelerators have received less attention. Alcantara et al. [2] were among the first to investigate hash map construction on CUDA-capable GPUs. Their single-GPU implementation employs a two-stage hashing cascade where keys are separated into buckets of size 512 residing in global memory. Subsequently, the buckets are rehashed in parallel using a third degree cuckoo hashing scheme which is performed on fast shared memory. In later work [7], the same authors propose a single-pass variant (*GPU cuckoo hash*) based on fourth degree cuckoo hashing which supports load factors of roughly 80% achieving an insertion performance of up to 250 million inserts per seconds on a GTX 470. In contrast, our WarpDrive probing scheme features significantly higher insertion rates even for load factors of more than 95%. Furthermore, WarpDrive can be scaled transparently over multiple GPUs.

García et al. proposed a GPU hash map based on Robin Hood Hashing [8] which equalizes probing lengths by augmenting each key with an additional 4-bit age indicator. Their implementation uses one thread for the insertion of a key-value pair in a lock-free manner at comparable speed to Alcantara's hash map. In contrast, our implementation utilizes a warp or a coalesced subgroup, respectively. As we will show, subwarp-synchronous probing is key to improve single-GPU performance. Note that WarpDrive is in principle amenable to Robin Hood Hashing based probing schemes as well.

Stadium Hash [9] uses an auxiliary table (ticket board) in addition to the hash table. Each CUDA thread attempts to insert a key by initially querying the ticket board. Only if the availability bit of the probed ticket board slot indicates that the corresponding hash table bucket is still available, the key-value pair is actually inserted – otherwise it is re-hashed. If the full hash map can be kept in GPU global memory (*in-core*) the performance of Stadium Hash is 1.04x-1.19x faster than GPU cuckoo hash on a GTX780 GPU at a load factor of 80% on average. Furthermore, Stadium Hash reports an *out-of-core* implementation where only the ticket board is kept in GPU global memory while the full hash table is stored in host memory. Even though expensive accesses to host memory are reduced by means of the ticket board, the performance drops to around 100 million inserts per second due to slow PCIe transfers. In this paper we address the limitation to the hash table size imposed by the bounded amount of global memory on GPUs by proposing a mechanism that allows for the efficient distribution of hash maps over multiple GPUs within the same node instead of using slow host memory accesses. Note that our distribution mechanism is not specific to any particular probing scheme but can in principle be incorporated into any of the previous single-GPU approaches.

IV. Massively Parallel Hash Maps

In this section, we treat the construction of single-GPU and multi-GPU hash maps separately. Subsection IV-A discusses our implementation of WarpDrive's probing scheme in the context of *warp-synchronous programming* up to CUDA 8 and its extension to *cooperative groups* in combination with *independent thread scheduling* introduced in CUDA 9. Subsection IV-B demonstrates how to distribute our hash map over multiple GPUs attached to the same compute node. This involves efficient partitioning of key-value pairs

and asynchronous inter-GPU communication using different interconnection network topologies.

A. Single-GPU Hash Map Construction

Our probing scheme adheres closely to the traditional warp execution model of CUDA-enabled GPUs where 32 contiguous threads (a so-called *warp*) are executed simultaneously on a Streaming Multiprocessor (SM) in lock-step manner. All threads within a warp perform the same instruction at the same time similar to the SIMD paradigm. The CUDA programming model relaxes the strict SIMD model by allowing distinct threads in the same warp to access non-contiguous memory or diverge into different execution branches at the cost of serialization. This computation model was coined by NVIDIA as SIMT (*Single Instruction Multiple Threads*). Although SIMT is more flexible, introducing non-coalesced memory accesses and branch-divergence often leads to performance degradation compared to strictly homogeneous SIMD algorithms.

Note that with the introduction of the Volta generation and CUDA 9, consecutive threads within a warp can be scheduled independently and thus have to be synchronized explicitly. Nevertheless, WarpDrive is applicable to both thread scheduling paradigms: traditional *warp-synchronous programming* on pre-Volta hardware and *independent thread scheduling* on (post-)Volta devices. At this point, let us fix the notation in order to comply with both programming paradigms: $|g| \in \{1, 2, 4, 8, 16, 32\}$ consecutive threads in the same thread block shall be denoted as *coalesced group* (CG) g. The special case $|g| = 32$ refers to a traditional warp. CGs are always implicitly synchronized on pre-Volta hardware but are not guaranteed to be executed in lock-step on (post-)Volta GPUs. Hence, we will always state a synchronization call using either the explicit `__syncwarp()` instruction or implicit synchronization issued by member functions of a CG.

Our approach implements an open addressing hash map based on a hybrid probing scheme that combines simultaneous linear probing within a batch consisting of $|g|$ slots and sequential chaotic probing of those batches. Parallel insertion is accomplished as illustrated in Fig. 2 and listed as pseudocode in Fig. 3. Each CG g independently inserts a key-value pair $d = (k, v)$ into the hash table t with capacity $c = |t|$. The parameter p_{max} corresponds to the outer probing loop (Line 4) and denotes the maximum number of unsuccessful probing attempts of g before raising an insertion error (Line 26). The inner probing loop (Line 6) ensures a consistent probing scheme in case that the size of g is varied over time. After applying a hash function to the key of d (Line 5) the resulting hash value h is used i) to determine the table slot to be probed by each thread in g (Line 7) and ii) to load the corresponding element d_t into fast registers (Line 8). The CG probes $|g|$ consecutive slots which results in a coalesced and therefore efficient global memory access pattern. At this certain point, the copies of the keys in registers might have already been deprecated since another CG may have occupied an empty or deleted slot in the same region. As a consequence, we have to guard the to be accomplished insertion with a CAS operation in a later phase. In the following, all members of the CG determine whether their stored key is an empty placeholder or tombstone (in case of previous deletions). The resulting binary mask is broadcasted as packed $|g|$-bit integer to all active members of the CG (Line 9). This can be accomplished with the intrinsic instruction `auto mask=__ballot[_sync](predicate)` which is either stated as explicit call in pre-Volta code or is a dedicated member function of a CG in (post-)Volta implementations. Let us assume that there is at least one unoccupied slot in that region (i.e. $mask \neq 0$), then we attempt to insert the key-value pair d at the leftmost position in the CG using an atomic CAS operation (Line 13). If successful, i) all group members are notified using the collective group predicate `__any[_sync](predicate)` or its CG member counterpart (Line 17) and ii) finally exit the function (Line 18). In case of failure, we consecutively probe the remaining active bits in ascending order until we exhaust the whole probing window (as illustrated in Fig. 2). If insertion remains unsuccessful even after successive reloading from global memory (Line 20), we move on to the next probing window. Queries are performed in a similar way whereby the atomic swap is not required. Note that the described pattern is safe in case of concurrently issued insertions and queries but cannot be used in combination with deletions. Nevertheless, insertions and deletions can be safely interleaved using global barriers. Also note that the collectives `g.ballot(predicate)` and `g.any(predicate)` implicitly synchronize all threads in a coalesced group.

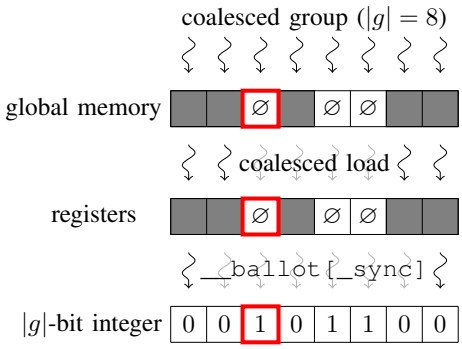

Fig. 2. Linear probing within a CG consisting of $|g| = 8$ contiguous threads. The thick red box denotes the leading thread which attempts the first probe.

B. Multi-GPU Hash Map Construction

Hash maps can be distributed in several ways over $m > 1$ GPUs. Distinct approaches can be categorized by their distribution patterns of key-value pairs. We use a a partition (hash) function $p(k) \in \{0, 1, \ldots, m-1\}$ which assigns each $k \in K$ a unique GPU identifier.

- **Host-sided partitioning:** key-value pairs are partitioned on the host by means of the value of $p(k)$. Subsequently, the partitions are transferred and inserted into their respective hash map residing on the corresponding GPU.

```
1: function INSERT(d, g, t, p_max)
2:     rank ← g.thread_rank
3:     success ← false
4:     for p ← 0, p_max do                          ▷ outer probing loop
5:         h ← hash(d, p)
6:         for q ← 0, 32/|g| do                     ▷ inner probing loop
7:             i ← (h + q · |g| + rank)  mod |t|
8:             d_t ← t[i]
9:             mask ← g.ballot(d_t = ∅)             ▷ sync
10:            while mask ≠ 0 do
11:                leader ← __ffs(mask)             ▷ leftmost active
12:                if rank = leader then
13:                    if d_t = CAS(t + i, d_t, d) then
14:                        success ← true
15:                    end if
16:                end if
17:                if g.any(success) then           ▷ sync
18:                    return                       ▷ insert successful
19:                else
20:                    d_t ← t[i]                   ▷ reload d_t
21:                    mask ← g.ballot(d_t = ∅)     ▷ sync
22:                end if
23:            end while
24:        end for
25:    end for
26:    raise insertion error
27: end function
```

Fig. 3. This function attempts to inserts a key-value pair d with at most p_{max} chaotic probes into the hash table t residing in global memory using a coalesced thread group g. Further details are provided in Section IV-A.

- **System-wide lock-free insertion:** the hash map resides simultaneously on all GPUs realized by unified memory addressing. Lock-free CAS instructions have to be issued across multiple devices using slow system-wide atomics.
- **Unstructured distribution:** key-value pairs are split into portions of approximately equal size and transferred to the GPUs. Each GPU maintains an independent hash map. Unfortunately, this implies that we have no a priori information on which device a specific key is stored.
- **Distributed multisplit transposition:** key-value pairs are initially transferred to m GPUs similar to unstructured distribution. Subsequently, they are separated on each GPU into m portions by means of $p(k)$. Afterwards, the $m \times m$ partitions are reshuffled across devices such that GPU i exclusively holds keys where $p(k) = i$.

Trivial *host-sided partitioning* can be ruled out from the very beginning since linear time reordering of elements in host RAM is almost as expensive as CPU-based hash map construction. Although solutions based on *system-wide atomics* are preferable in terms of code complexity, they tend to be unreasonably slow in our preliminary experiments. *Unstructured distribution* avoids host-sided reordering of elements and is only bounded by the bandwidth of asynchronous memory transfers from the host to the devices. Nevertheless, querying is cumbersome and time-consuming since we have no a priori information about the location of a certain key. *Distributed multisplit transposition* avoids the brute-force querying of all devices since each GPU exclusively holds a subset of a priori known keys. The involved partitioning (multisplit) into m classes can be accomplished in fast video RAM in contrast to slow *host-sided partitioning* in ordinary RAM. The resulting $m \times m$ partition table (m partitions on m GPUs) is transposed in a subsequent step by communicating $m - 1$ partitions from each of the m GPUs to their respective target device. *All-to-all* communication is bounded by the overall bandwidth of the utilized interconnection network topology. As we will show, both the concurrent multisplit computation and subsequent transposition can be performed efficiently on multi-GPU nodes with NVLINK support.

Single-GPU multisplit could be performed by sorting key-value pairs according to the value of $p(k)$ using massively parallel radix sort as provided by CUB [17]. However, Ashkiani et al. [22] proved that the same can be accomplished with less computational effort. Their CUDA implementation computes a histogram consisting of m slots using a hierarchy of register-based shuffles in a warp, shared memory-based shuffles in a thread block, and global memory-based shuffles over the whole device. Our approach is based on a simpler technique that consecutively computes m binary splits (one class versus the rest) of keys in global memory. This can be accomplished using a warp-aggregated atomic counter that increments the final position of a key within a coalesced group as described in [23]. Although warp-aggregated compression is slightly slower than Ashkiani's full stack GPU multisplit implementation, we stick to our basic approach. It only accounts for a minor portion of the overall runtime and thus further optimization does not significantly increase performance.

The resulting $m \times m$ partition table $T[gpu, part]$ stores the number of elements and associated pointers of each partition $part$ residing on GPU gpu. In the following step, we transpose T by asynchronously sending $m^2 - m$ off-diagonal elements ($gpu \neq part$) to their corresponding target device. The resulting key-value distribution $T^t[part, gpu]$ concatenates all elements belonging to a certain partition identifier $p(k) = part$ originally stemming from GPU gpu on the same device. Offsets are computed using row-wise exclusive prefix scans over T for the senders and column-wise scans for the receivers. The combination of both intra-device multisplit and subsequent cross-device transposition is equivalent to a distributed multi-GPU multisplit primitive. Distributed insertion is realized with a *multisplit* $\rightarrow$ *transposition* $\rightarrow$ *insert* cascade. Note that matrix transposition is an isomorphism and thus all-to-all communication is reversible as well. Hence, querying can be accomplished using a *multisplit* $\rightarrow$ *transposition* $\rightarrow$ *query* $\rightarrow$ *transposition* cascade. Fig. 4 illustrates the workflow.

The overall performance of distributed insertion and querying depends on the bandwidth of four basic operations. First, asynchronous host-to-device communication is accomplished over a bus (usually PCIe, infrequently NVLINK) to the devices and vice versa. Hence, we are limited to a few tens of

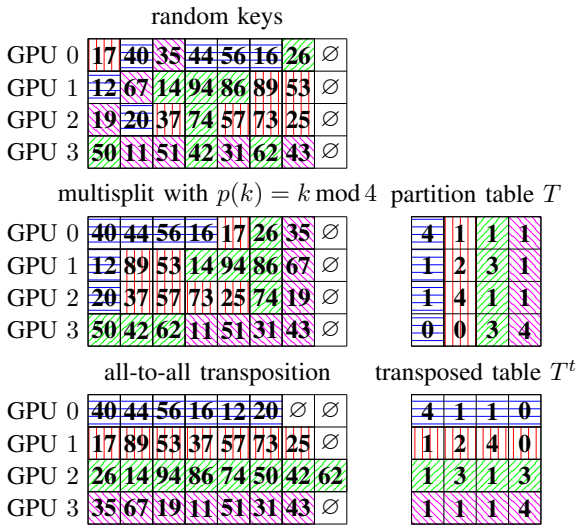

Fig. 4. Example for distributed multisplit of $4 \times 7 = 28$ random (not necessarily unique) keys over four GPUs. Colored patterns correspond to the value of the partition hash function $p(k) = k \mod 4 : 0 \mapsto$ blue horizonal lines, $1 \mapsto$ red vertical lines, $2 \mapsto$ green north east lines, $3 \mapsto$ magenta north west lines. The upper panel shows the initial setting. The centered panel depicts the key distribution and associated partition table T after independently performing a single-GPU multisplit on each GPU. The final transposition of the partition table is achieved with an all-to-all communication pattern. Note that all operations are issued out-of-place using one double buffer per GPU of sufficient size (8 in this example).

gigabytes per second in case of PCIe which is a bottleneck even for single-GPU hash maps. Fortunately, this step can often be bypassed if the to be inserted data already resides on the GPUs in case of a tight integration into a processing tool chain or alternatively can be generated on-the-fly. As an example, bioinformatics applications often extract and hash all $n - k + 1$ substrings of length k (called k-mers) from a DNA sequence of length n where $k \leq n$. Thus, keys of overall size $k \cdot (n - k + 1) \in \mathcal{O}(n \cdot k)$ can be generated on the devices from only $\mathcal{O}(n)$ data that has been transferred to the GPUs in a previous step. In this case, the effective transfer rate over the PCIe bus is artificially increased by a factor of approximately k. Other examples include windowed patch extraction from images or high-resolution rasterization of 3D meshes. Second, the single-GPU multisplit primitive is exclusively executed on fast video RAM and thus has only minor impact on the performance. Third, the all-to-all transposition step communicates data among the GPUs. Consequently, transposition speed heavily depends on the bandwidth of the interconnection network topology. Fourth, insertion and querying of key-value pairs is exclusively performed on fast video RAM. Unfortunately, we can only saturate a fraction of the overall bandwidth due to the random nature of hashing.

For the sake of simplicity, let us make the (realistic) assumption that the time needed for PCIe transfers is similar to the accumulated time needed for the multisplit, transpose, and insertion steps. In this case, the overall performance degrades to half the PCIe bandwidth since the whole traversal of the insertion cascade relies on global barriers. Nevertheless, PCIe transfers issued in one CPU thread on the host can be overlapped with the execution of the remaining communication and insertion primitives of another batch issued in a second multi-thread. The same is true for exclusively device-sided primitives: partitioning of data on GPUs can be overlapped with all-to-all communication and so forth. Our WarpDrive implementation supports asynchronous insertion and querying with a user-defined number of CPU threads in order to fully utilize the available hardware resources (see Figure 5).

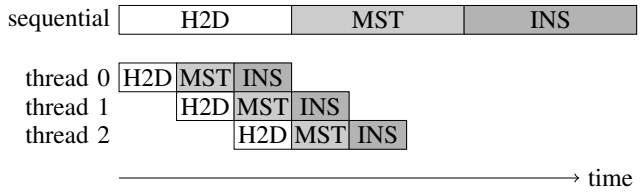

Fig. 5. Example demonstrating the efficient overlapping of host-to-device memory transfers (H2D), multisplit + transposition (MST), and insertion (INS) using 3 multi-threads. The H2D $\rightarrow$ MST $\rightarrow$ INS cascade remains sequential within each batch. However, the runtime of distinct primitives in different threads can be overlapped. Note that each block within a batch utilizes different hardware resources of a multi-GPU node (H2D: PCIe bus, MST: mainly NVLINK interconnection network, INS: video memory).

V. Experiments

A. Experimental Setup

All experiments are conducted on a multi-GPU compute node being part of the Mogon II supercomputer at JGU Mainz.

- **CPU**: dual socket Intel Xeon E5-2680 v4@2.40GHz featuring 2×14 physical cores with hyperthreading support.
- **RAM**: 16×16 GB $= 256$ GB of DDR4 modules (ECC).
- **GPU**: 4 NVIDIA Tesla P100 boards@1.48GHz.
- **VRAM**: 16 GB of HBM2 stacked memory for each GPU featuring up to 720 GB/s peak bandwidth.
- **NVLINK**: augmented fully connected graph consisting of 4×4 bidirectional links with 20 GB/s bandwidth each.
- **Software**: CUDA 9.0, GCC 5.4.0, CentOS Linux 7

A schematic layout of the NVLINK based GPU interconnection network topology is illustrated in Fig. 6. At least one bidirectional NVLINK edge is established between each pair of GPUs. Additionally, two parallel edges of the 2D hypercube subnetwork are augmented with an additional edge. Each pair of GPUs is connected via a dedicated PCIe switch to a CPU. Note that this corresponds to an accumulated theoretical peak bandwidth for asynchronous host-to-device transfers of 24 GB/s ($\approx$ 22 GB/s in experiments).

Performance evaluation is conducted on three distinct 4-byte key distributions with arbitrary 4-byte values.

- **Unique distribution**: up to 2^{32} unique keys are sampled without replacement from the space of 4-byte keys. This sampling method is equivalent to a Fisher-Yates shuffle of an ascending integer sequence.

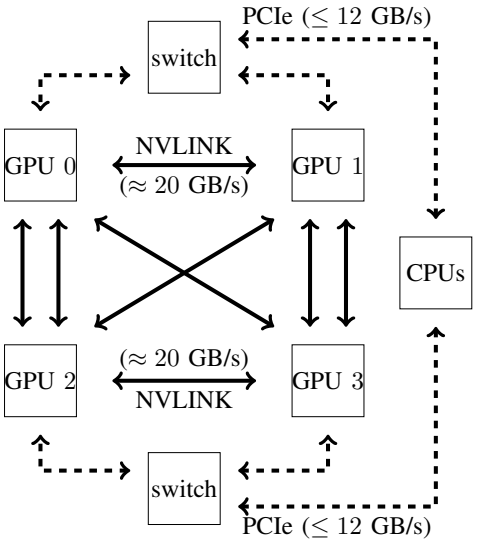

Fig. 6. Interconnection network of a multi-GPU node with 4 Tesla P100 devices. NVLINK supports bidirectional access between GPUs featuring ≈ 20 GB/s bandwidth per NVLINK edge. Host-to-device traffic is routed with $\leq 2 \times 12$ GB/s bandwidth over 2 PCIe switches.

- **Uniform distribution**: up to 2^{32} keys are drawn with replacement from a uniform distribution. The number of unique keys scales with the bootstrap ratio $(1-e^{-n/2^{32}})$.
- **Zipf distribution:** the multiplicity of keys is distributed according to a power law. The multiplicity of a key with rank k is smaller than the one of the most common key by a factor of k^{-s} where $s > 1$ is an exponential damping coefficient [24].

The quotient of the number of inserted elements and the capacity (target load factor) coincides with the true load factor α for unique keys but may overestimate α for the remaining distributions.

The employed 4-byte hash functions are either the integer finalizer of Appleby's popular MurmurHash3 implementation [25] or a similar approach proposed by Mueller [26]. Both functions exhibit favorable avalanche properties and further act as isomorphism on the space of 4-byte integers (being index permutations). Hence, translated variants $\tilde{h}_y(x) = h(x + y)$ for $x, y \in$ `uint32_t` sustain the mathematical properties.

```
// murmur integer finalizer
uint32_t fmix32 (uint32_t x)
{
    x ^= x >> 16;
    x *= 0x85ebca6b;
    x ^= x >> 13;
    x *= 0xc2b2ae35;
    x ^= x >> 16;
    return x;
}
```

```
// mueller hash
uint32_t mueller(uint32_t x)
{
    x ^= x >> 16;
    x *= 0x45d9f3b;
    x ^= x >> 16;
    x *= 0x45d9f3b;
    x ^= x >> 16;
    return x;
}
```

Execution times are reported as averaged wall clock time over several runs. Time measurement is accomplished with the help of precise timers provided by the CUDA event system. In all experiments, we assume that the to be inserted or retrieved data resides either in host RAM for operations issued from the CPU or in video RAM for device-sided benchmarks. Test data generation and the loading of files from disk are omitted.

B. Single-GPU Performance

In this subsection we discuss the performance of WarpDrive on a single Tesla P100 GPU using different input distributions and load factors. Additionally, we compare WarpDrive to Alcantara's single GPU implementation [2] which is included in the *CUDA Data Parallel Primitives Library (CUDPP)*. To our best knowledge, CUDPP is the only publicly available implementation amongst competing GPU hash tables.

The experimental protocol reads as follows: For each input distribution we i) insert 2^{27} (4+4)-byte key-value pairs (1 GB) residing in video memory into the hash table and subsequently ii) retrieve all elements to video memory. For both tasks we measure the kernel execution time for different load factors ranging from 40% to 99%. Note that CUDPP is constrained to a maximum load of 97%. The key distributions are those mentioned in Section V-A. However, uniformly drawing 2^{27} keys out of 2^{32} unique elements with replacement results in a proportion of $\left(1-\exp(-\frac{2^{27}}{2^{32}})\right)\frac{2^{32}}{2^{27}} \approx 98.5\%$ unique keys and is therefore almost indistinguishable from a purely unique key set in terms of insertion/retrieval performance. Hence we omit the evaluation for this distribution for the given configuration.

Figure 7 illustrates the insertion/retrieval performance in billion operations per second for a unique random key set. The results show that the performance of WarpDrive (WD) highly depends on the chosen group size $|g|$. A large group size increases the probability of finding an unoccupied slot within the given probing window, whereas small groups may probe multiple windows at a higher group occupancy rate on the Streaming Multiprocessors. This trade-off is applicable to both insertion and retrieval. With increasing load larger group sizes get more favorable but optimal performance is achieved with $|g| \in \{2, 4, 8\}$. WarpDrive shows speedups over CUDPP of 1.79, 2.18, 2.84 for insertion and 1.3, 1.34, 1.3 for retrieval at load factors of 0.8, 0.9, 0.95 respectively. Note that the case $|g| = 1$ represents the naïve approach where one data element is processed by a single hardware thread. Unlike on previous architectures this approach is competitive to CUDPP on a Tesla P100 for reasonable loads.

The results of the same experiment under a Zipf distributed key set ($s = 1 + 10^{-6}$) are shown in Figure 8. Multiple elements share the same key due to the nature of the underlying distribution and thus share the same table slot. CUDPP does not support key collisions unless a multi-value hash table is used. However, our implementation resolves such collisions by updating an already written value for a colliding key. Hence, the value associated to a non-unique key is the last element written on the event horizon of the insertion kernel for this key. The observations made from the unique distribution also hold for the Zipf distribution but even smaller group sizes are favorable. Note that in this case the specified loads refers to the actual occupancy of table slots after inserting all elements.

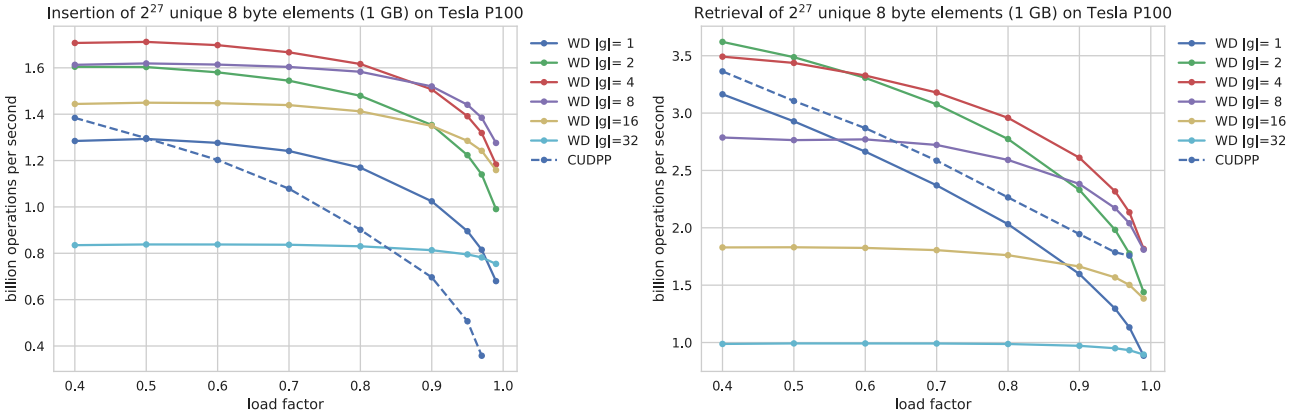

Fig. 7. Device-sided insertion and retrieval rates for varying group size parameters and load factors: unique distribution.

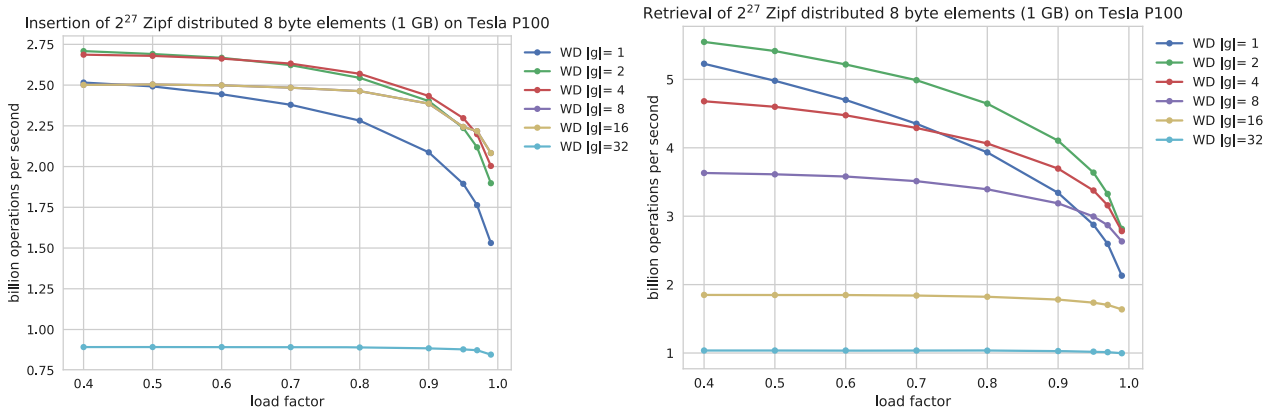

Fig. 8. Device-sided insertion and retrieval rates for varying group size parameters and load factors: Zipf distribution.

C. Multi-GPU Performance

This subsection investigates the weak and strong scaling behavior of our proposed multi-GPU distribution cascades for insertion and retrieval. Moreover, we shed light on the impact of asynchronous overlapping techniques. Due to space restriction we focus on a subset of possible hash map configurations: a reasonably fast but not optimal setting uses a coalesced group size of $|g| = 4$ and a target load factor of 95% for values of n ranging from 2^{28} (2 GB) to 2^{32} (32 GB) packed 64-bit key-value pairs. Insertion and retrieval operations are performed in batches consisting of 2^{24} elements (128 MB).

Initially, we perform a strong and weak scaling analysis on unique keys for both device-sided cascades: i) multisplit $\rightarrow$ transposition $\rightarrow$ insertion, and ii) multisplit $\rightarrow$ transposition $\rightarrow$ retrieval $\rightarrow$ transposition. The experimental setup reads:

- **Strong scaling**: overall $n \in \{2^{28}, 2^{29}\}$ packed pairs (2 GB / 4 GB) are inserted/retrieved into/from m GPUs. As a result, each GPU processes $\frac{n}{m}$ elements in parallel.
- **Weak scaling**: $n \in \{2^{28}, 2^{29}\}$ packed pairs (2 GB / 4 GB) are inserted/retrieved into/from each of the m GPUs. Hence $m \cdot n$ elements are processed overall in parallel ranging from $1 \cdot 2^{28}$ (2 GB) to $4 \cdot 2^{29}$ (16 GB) pairs.

Let $\tau(n, m)$ be the time needed to process n elements on m GPUs then strong and weak scaling efficiency are defined as

$$E_{\text{s}}(n, m) = \frac{\tau(n, 1)}{m \cdot \tau(n, m)}, \; E_{\text{w}}(n, m) \quad = \frac{\tau(n, 1)}{\tau(m \cdot n, m)} \tag{4}$$

The results are shown in Fig. 9. Both the strong and weak scaling efficiency remain constant for $m \geq 2$ which implies good scalability. The efficiency drop from $m = 1$ to $m = 2$ can be explained by the time needed for the additional multi-split and communication primitives. The super-linear speedup observed in the strong scalability analysis during the insertion of 2^{29} elements is caused by performance degradation of the single-GPU implementation for increasing capacities. A detailed explanation is provided in the following experiment.

We further investigate the scalability in terms of varying capacities for $m = 4$ GPUs using both the aforementioned device-sided cascades and two additional host-sided insertion and retrieval cascades. The latter prepend or append PCIe transfers between host and devices to the already existing device-sided pipelines. The experiments include the insertion and retrieval of 2^{28} (2 GB) to 2^{32} (32 GB) elements for three distinct key distributions (unique, uniform, Zipf: $s = 1+10^{-6}$). Fig. 10 depicts the associated insertion and retrieval rates. Query performance remains constantly high at up to 9

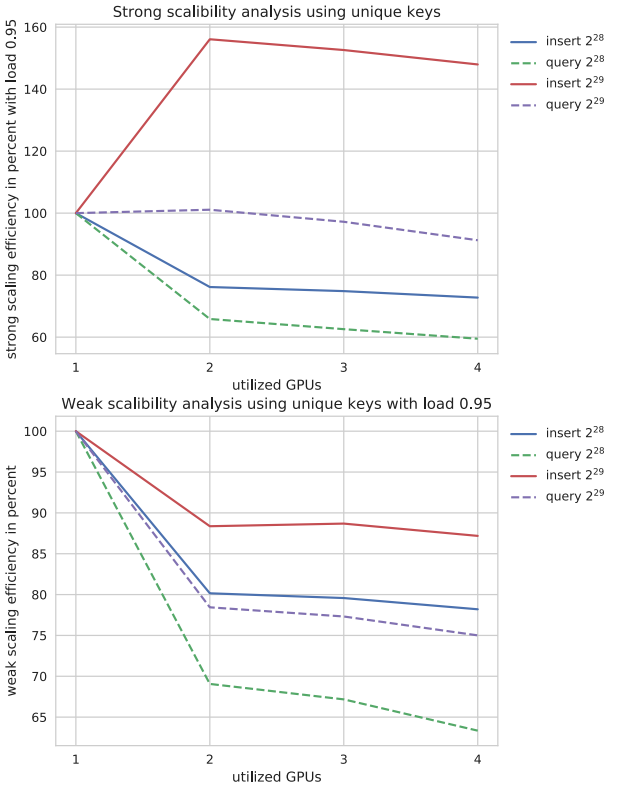

Fig. 9. Strong and weak scalability analysis using 1, 2, 3, and 4 GPUs. The super-linear strong speedup of 'Insert 2^{29}' (red solid line in upper panel) can be explained by performance degradation of the single-GPU implementation for an increasing number of inserted elements n with a load of $\alpha = 95\%$.

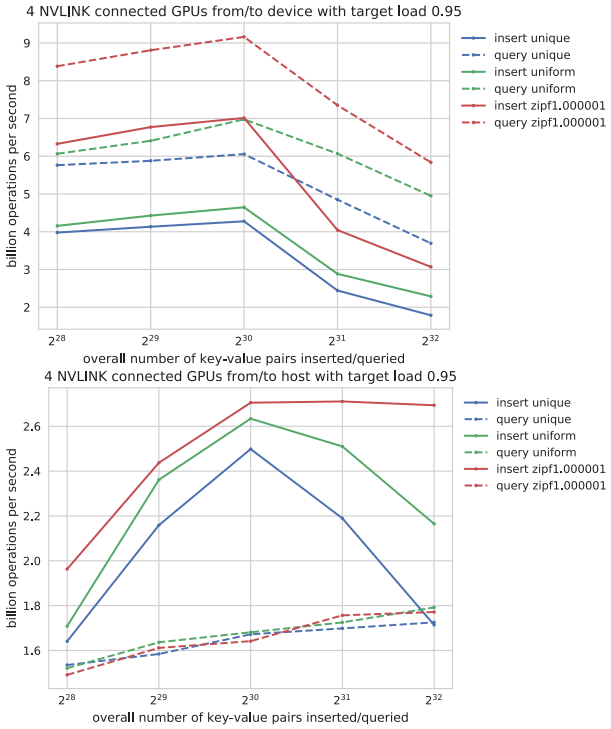

Fig. 10. Insertion and retrieval performance for distinct key distributions without PCIe transfers (upper panel) and including memory transfers from/to the host (lower panel). Note that retrieval performance for the host variant (dashed lines in lower panel) involves two sequentially issued PCIe transfers: i) copying the keys from the host to the GPUs and ii) transferring the resulting key-value pairs back to the host.

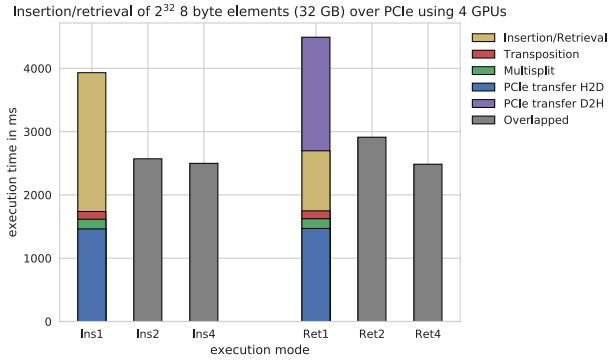

Fig. 11. Runtime decomposition of insertion and retrieval cascades on an NVLINK-enabled node including PCIe transfers for 32 GB of 8 byte key-value pairs. Ins1 and Ret1 denote the sequentially issued insertion and retrieval cascades. Ins2/Ins4 and Ret2/Ret4 correspond to the multi-threaded variants that overlap 2 or 4 cascades, respectively. The accumulated execution times can be significantly reduced for Ins2, Ins4, Ret2, and Ret4.

billion operations per second on the device over all tested distributions and capacities. However, device-sided insertion performance drops by up to a factor of two for $n > 2^{30}$ elements (> 2 GB on each of the 4 GPUs). The 16 GB video memory of a P100 GPU are addressed via 8 memory interfaces. Hence, we suspect that atomic CAS might degrade if lock-free instructions are issued across several memory interfaces. Note that this artifact explains the super-linear speedup in the aforementioned strong scaling benchmark. Host-sided insertions are faster than queries since the retrieval cascade involves an additional PCIe transfer. Nevertheless, the peak insertion/retrieval rates from/to the host correspond to 84%/55% of the theoretically achievable PCIe bandwidth. As a result, host-sided insertion is comparably fast as plain memcopies from RAM to global memory.

Finally, we shed light on the impact of asynchronous cascade overlapping. Fig. 11 illustrates the runtime decomposition of two sequentially issued insertion and retrieval cascades over PCIe together with their asynchronous variants using 2 and 4 multithreads. The execution times of the overlapped variants can be reduced by up to 36% for insertion, and 45% for querying in comparison to their sequential counterparts. The fractions of multisplit and transposition range between 2% and 4% of the overall execution time. In detail, multisplit performs at ≈ 210 GB/s accumulated bandwidth on global memory and all-to-all transposition corresponds to ≈ 192 GB/s bandwidth of the NVLINK interconnection network.

VI. Conclusion

Hash maps are of high importance to a variety of applications. In this paper, we have introduced WarpDrive – an

efficient multi-GPU hash map implementation targeting single compute nodes. WarpDrive features three improvements over the state-of-the-art:

1) a GPU-based hash map algorithm featuring a novel subwarp-level probing scheme tailored to suit the characteristics of high latency and high bandwidth video memory.
2) a scalable multi-GPU hash map construction/querying scheme that allows the building of hash maps containing several billions of key-value pairs by exploiting fast GPU interconnection networks.
3) an efficient asynchronous technique that allows for the overlapping of communication primitives and insertion/retrieval kernels.

The single-GPU variant performs at up to $\approx (1.7-2.7)\times 10^9$ operations per second for device-sided insertion and $\approx (3.5-5.5)\times 10^9$ operations per second during device-sided retrieval. The multi-GPU implementation utilizing four CUDA accelerators provides device-sided insertion rates up to $\approx (4-7)\times 10^9$ operations and $\approx (6-9)\times 10^9$ queries per second. Host-sided operations including PCIe transfers can be accomplished at up to $\approx (2.5-2.7)\times 10^9$ insertions and $\approx 2\times 10^9$ queries per second. Moreover, we have proposed a mutli-GPU multisplit primitive that allows for the efficient distribution of key-value pairs across up to four GPUs. Both building blocks (intra-GPU multisplit and inter-GPU transposition) process approximately 200 GB/s which is key for scalability.

Our experiments reveal that hash map performance is heavily influenced by a non-trivial relationship between the three parameters: load factor α, group size $|g|$, and capacity c. A possible direction for future research could be design of a heuristic which dynamically scales the group size $|g|$ with the current load factor. Furthermore, we have observed that single-GPU performance decreases gradually for capacities $c > 2$ GB. A possible workaround to further increase performance could be the partitioning of high capacity hash maps into several smaller hash maps each of size ≤ 2 GB.

ACKNOWLEDGMENT

Parts of this research were conducted using the supercomputer Mogon II and/or advisory services offered by Johannes Gutenberg University Mainz (hpc.uni-mainz.de) which is a member of the AHRP and the Gauss Alliance e.V.

We further would like to thank Computational Science Mainz (CSM) for additional funding.

REFERENCES

[1] C.-Y. Lai, "Efficient Parallelization of Natural Language Applications using GPUs," Master's thesis, EECS Department, University of California, Berkeley, May 2012.

[2] D. A. Alcantara, A. Sharf, F. Abbasinejad, S. Sengupta, M. Mitzenmacher, J. D. Owens, and N. Amenta, "Real-time Parallel Hashing on the GPU," in *ACM SIGGRAPH Asia 2009 Papers*, ser. SIGGRAPH Asia '09. New York, NY, USA: ACM, 2009, pp. 154:1–154:9.

[3] J. Pan and D. Manocha, "Fast GPU-based Locality Sensitive Hashing for K-nearest Neighbor Computation," in *Proceedings of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, ser. GIS '11. New York, NY, USA: ACM, 2011, pp. 211–220.

[4] R. Ounit, S. Wanamaker, T. J. Close, and S. Lonardi, "CLARK: fast and accurate classification of metagenomic and genomic sequences using discriminative k-mers," *BMC Genomics*, vol. 16, no. 1, p. 236, Mar 2015.

[5] R. Kobus, C. Hundt, A. Müller, and B. Schmidt, "Accelerating metagenomic read classification on CUDA-enabled GPUs," *BMC Bioinformatics*, vol. 18, no. 1, p. 11, Jan 2017.

[6] A. M. Aji, L. Zhang, and W. c. Feng, "GPU-RMAP: Accelerating Short-Read Mapping on Graphics Processors," in *2010 13th IEEE International Conference on Computational Science and Engineering*, Dec 2010, pp. 168–175.

[7] D. A. F. Alcantara, "Efficient Hash Tables on the GPU," Ph.D. dissertation, Davis, CA, USA, 2011, aAI3482095.

[8] I. García, S. Lefebvre, S. Hornus, and A. Lasram, "Coherent Parallel Hashing," in *Proceedings of the 2011 SIGGRAPH Asia Conference*, ser. SA '11. New York, NY, USA: ACM, 2011, pp. 161:1–161:8.

[9] F. Khorasani, M. E. Belviranli, R. Gupta, and L. N. Bhuyan, "Stadium Hashing: Scalable and Flexible Hashing on GPUs," in *2015 International Conference on Parallel Architecture and Compilation, PACT 2015, San Francisco, CA, USA, October 18-21, 2015*, 2015, pp. 63–74.

[10] T. Maier, P. Sanders, and R. Dementiev, "Concurrent Hash Tables: Fast and General?(!)," in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '16. New York, NY, USA: ACM, 2016, pp. 34:1–34:2.

[11] M. M. Michael and M. L. Scott, "Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms," Rochester, NY, USA, Tech. Rep., 1995.

[12] A. Pagh, R. Pagh, and M. Ruzic, "Linear Probing with Constant Independence," in *Proceedings of the Thirty-ninth Annual ACM Symposium on Theory of Computing*, ser. STOC '07. New York, NY, USA: ACM, 2007, pp. 318–327.

[13] M. Thorup and Y. Zhang, "Tabulation-Based 5-Independent Hashing with Applications to Linear Probing and Second Moment Estimation," *SIAM J. Comput.*, vol. 41, no. 2, pp. 293–331, Apr. 2012.

[14] P. G. Bradford and M. N. Katehakis, "A Probabilistic Study on Combinatorial Expanders and Hashing," *SIAM J. Comput.*, vol. 37, no. 1, pp. 83–111, Apr. 2007.

[15] P. Celis, P. A. Larson, and J. I. Munro, "Robin Hood Hashing," in *26th Annual Symposium on Foundations of Computer Science (sfcs 1985)*, Oct 1985, pp. 281–288.

[16] M. Herlihy, N. Shavit, and M. Tzafrir, "Hopscotch Hashing," in *Proceedings of the 22Nd International Symposium on Distributed Computing*, ser. DISC '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 350–364.

[17] D. Merrill and NVIDIA-Labs, "CUDA UnBound (CUB) Library." [Online]. Available: https://nvlabs.github.io/cub/

[18] D. Knuth, "Notes On "Open" Addressing," 1963.

[19] Y. Matias and U. Vishkin, "Converting High Probability into Nearly-constant Time – with Applications to Parallel Hashing," in *Proceedings of the Twenty-third Annual ACM Symposium on Theory of Computing*, ser. STOC '91. New York, NY, USA: ACM, 1991, pp. 307–316.

[20] J. Preshing, "Junction Concurrent Hash Map." [Online]. Available: http://preshing.com/20160201/new-concurrent-hash-maps-for-cpp/

[21] X. Li, D. G. Andersen, M. Kaminsky, and M. J. Freedman, "Algorithmic Improvements for Fast Concurrent Cuckoo Hashing," in *Proceedings of the Ninth European Conference on Computer Systems*, ser. EuroSys '14. New York, NY, USA: ACM, 2014, pp. 27:1–27:14.

[22] S. Ashkiani, A. Davidson, U. Meyer, and J. D. Owens, "GPU Multisplit," in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '16. New York, NY, USA: ACM, 2016, pp. 12:1–12:13.

[23] A. Adinetz and NVIDIA, "NVIDIA devblog: warp-aggregated atomics." [Online]. Available: https://devblogs.nvidia.com/parallelforall/cuda-pro-tip-optimized-filtering-warp-aggregated-atomics/

[24] L. A. Adamic and B. A. Huberman, "Zipf's law and the Internet," *Glottometrics*, vol. 3, pp. 143–150, 2002. [Online]. Available: http://www.hpl.hp.com/research/idl/papers/ranking/adamicglottometrics.pdf

[25] A. Appleby, "MurmurHash3 as part of SMHasher." [Online]. Available: https://github.com/aappleby/smhasher/

[26] T. Mueller, "32 bit and 64 bit Mueller Hash Functions." [Online]. Available: https://stackoverflow.com/a/12996028