Accelerating Parallel Operation for Compacting Selected Elements on GPUs

Johannes Fett, Urs Kober, Christian Schwarz, Dirk Habich, and Wolfgang Lehner

Technische Universität Dresden, Database Research Group, Germany
firstname.lastname@tu-dresden.de

Abstract. Compacting is a common and heavily used operation in different application areas like statistics, database systems, simulations and artificial intelligence. The task of this operation is to produce a smaller output array by writing selected elements of an input array contiguously back to a new output array. The selected elements are usually defined by means of a bit mask. With the always increasing amount of data elements to be processed in the different application areas, better performance becomes a key factor for this operation. Thus, exploiting the parallel capabilities of GPUs to speed up the compacting operation is of great interest. In this paper, we present different optimization approaches for GPUs and evaluate our optimizations (i) on a variety of GPU platforms, (ii) for different sizes of the input array, (iii) for bit distributions of the corresponding bit mask, and (iv) for data types. As we are going to show, we achieve significant speedups compared to the state-of-the-art implementation.

Keywords: Compacting · GPU · Optimization · Parallel

1 Introduction

A common observation in different application domains like statistics, database systems, simulations and artificial intelligence is that highly parallel algorithms in these domains usually produce sparse data or data containing unwanted elements [11, 16, 17]. To achieve a high performance for the following algorithm steps, it is often necessary to compact the data prior to these steps. The parallel breadth first tree traversal is one example [17]. Here, the list of open nodes must be pruned of invalid nodes after each traversal step. Otherwise, an exponential explosion of nodes takes place. A second example is the filter operation in database systems to reduce data based on predicates. These filters are usually executed as close to the base data as possible to reduce the effort of subsequent joins or groupings [9]. In all cases, the reduction of the data to the selected elements is performed with the *compaction* operation or primitive. This primitive is also denoted as stream compaction, stream reduction, or selection and the task

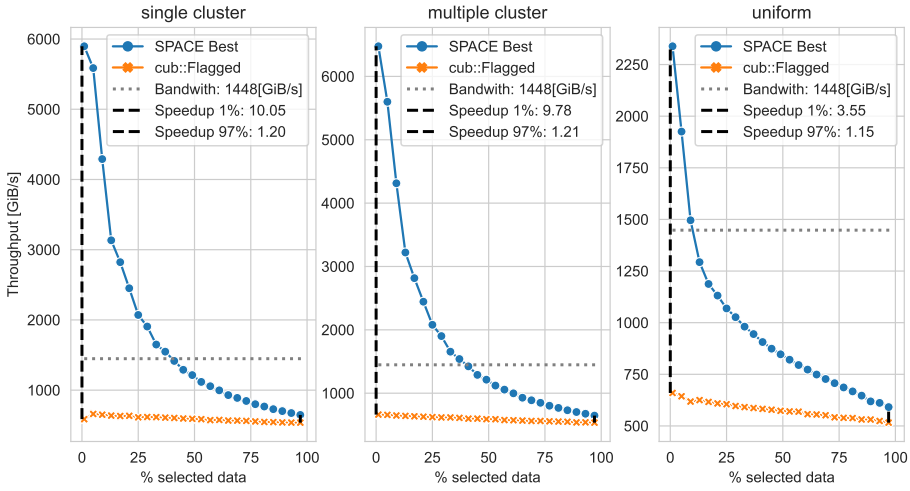

Fig. 1. *SPACE* compared to CUB on A100 Ampere GPU. Uniform, 1 cluster and 32 cluster data distributions for the bit mask. Datatype is unsigned integer 32-bit.

of this primitive is to produce a smaller output array by writing selected elements of an input array contiguously back to a new output array. The selected elements are usually defined by means of a bit mask. With the still increasing amount of data to be processed in the different application domains, high performance for this key primitive is a decisive factor.

In the last decade, GPUs have been increasingly used for highly parallel computations or accelerating specific algorithm parts [3, 8, 10, 13]. Thus, exploiting the parallel capabilities of GPUs to speed up the compacting operation is of great interest. For NVIDIA GPUs, the CUB library provides state-of-the-art, reusable software components for every layer of the CUDA programming model [5]. In particular, CUB also provides an efficient implementation for *compaction primitive* – called `cub::DeviceSelect:Flagged` – as highlighted in Figure 1. For the illustrated results, we generated a data array and three different bit mask configurations, we varied the percentage of selected elements and we executed the compaction primitive on these settings using a recent NVIDIA GPU A100. One bit mask configuration is called *single cluster* where the selected elements are contiguous in a single cluster. A second bit mask configuration is called *multiple cluster* where the selected elements are contiguous in several clusters and the clusters are uniformly distributed within the bit mask. And the third configuration is denoted as *uniform* because the selected elements are uniform distributed over the bit mask. As shown in Fig. 1, the CUB primitive provides a stably high throughput for all settings, but is not optimized in terms of the percentage of the selected items. That means, the state-of-the-art CUB implementation does not have any specialization for *compaction* with a very low percentage of selected data.

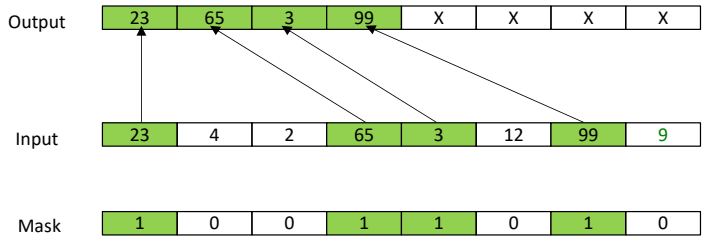

Fig. 2. Compaction example.

Our Contribution and Outline: However, as *compaction* only writes back selected data, low percentages of selected data offer a great opportunity for optimizations. Thus, we present in this paper several optimizations for this called *smart partitioning for GPU compaction (SPACE)* in this paper. As implied in Fig. 1, our best performing implementation clearly outperforms the state-of-the-art CUB implementation. The worst performing bit mask distribution uniform offers a throughput improvement of 3.55x for *SPACE* with 1 % of data selected. For 97 % selected data the improvement remains 1.15x. The best bit mask distribution is single cluster. In this case, *SPACE* achieves an improvement of 10.05x and a worst case improvement of 1.2x.

The remainder of this paper is structured as follow: In Section 2, we briefly introduce the necessary background. Then, we present our *smart partitioning for GPU compaction (SPACE)* approach in Section 3. Afterwards, we present selective results of our exhaustive evaluation. Finally, we conclude the paper with related work in Section 5 and a short summary in Section 6.

2 Preliminaries

In this section, we introduce all essential preliminary requirements for our work. We start with a clear description of the *compaction* primitive, followed by a classification of possible bit mask configurations and finally, we shortly recap the architecture of NVIDIA GPUs.

2.1 Compaction Primitive

Compaction is a common programming primitive that has a wide range of applications. As illustrated in Fig 2., the input of the primitive is an input array and a bit mask. The bit mask is used to indicate which elements should be selected. Thus, the number of bits in the bit mask is equal to the number of elements in the input array. Then, the primitive produces a new output array, containing only selected elements from the input data array, which are indicated by bit set to 1 in the bit mask. The most important challenge is that the selected elements

have to be written contiguously into the output array as shown in Fig. 2. In case of the NVIDIA GPUs, NVIDIA already ships a library called CUB that offers a performant implementation of the compaction based on a bit mask. The function is called `cub::DeviceSelect::Flagged`.

2.2 Bit Mask Characteristics

The bit mask as input essentially determines how many elements must be written to the output. If the percentage of set ones is small, then only a small amount needs to be written out. However, the percentage says nothing about where the set ones are in the bit mask. To be able to specify this more precisely, we examined various algorithms and determined the following: On the one hand, there are applications where the set ones are uniformly distributed in the bit masks. On the other hand, there are also applications where the set ones occur contiguously – clustered – in the bit mask. Especially in the case of clustered ones, large parts of the input array can actually be skipped, which can be used for optimizing the *compaction* primitive. However, our experiments from the introduction clearly show that the state-of-the-art CUB implementation does not have such an optimization.

2.3 NVIDIA GPU Architecture and Execution Model

The GPU architecture consists of a set of streaming multiprocessors. Each streaming multiprocessor consists of CUDA cores that are arithmetical logical units and exclusive local memory, which is called shared memory. A larger global memory can be accessed by all streaming multiprocessors. In order to perform calculations on a GPU, a set of threads is spawned and partitioned into work units called blocks. One block can have a maximum of 1024 threads and is assigned to one streaming multiprocessor. Multiple blocks can be assigned to a streaming multiprocessor. Instructions are executed for 32 threads at the same time, which is called a warp. Each streaming multiprocessor has a set of warp schedulers that schedule execution of different warp wide instructions. For example, Turing based GPUs have 4 warp schedulers per block and up to 4 warps can be scheduled at each unit to hide memory access latency. This leads to 16 warps or 512 threads as recommended threads per block [15]. While the basic architecture remains the same across GPU generations, there are differences in terms of memory speed, CUDA cores per streaming multiprocessor, shared memory per streaming multiprocessor and total streaming multiprocessors per GPU. As instructions are executed per warp ideally, multiple executions per warp are needed if branching or non-aligned memory access patterns occur.

The execution model is *Single Instruction Multiple Threads (SIMT)*. Unlike *Single Instruction Multiple Data (SIMD)* architectures like AVX512, where one instruction is executed on a vector of elements, GPU threads are able to behave independently from each other. Programs that use CUDA always consist of two parts of code. Host code runs on the CPU and is tasked with managing data transfers and execution of kernels. Device code runs on the GPU and consists

of kernel functions. Kernels are C-style functions that execute instructions on data, based on a `ThreadID`. Each kernel is called with a blocksize (number of threads within a block) and a gridsize (total number of blocks). Both parameters together are called a CUDA configuration. Setting the right CUDA configuration is critical to achieve good performance and varies between different kernels.

3 SPACE - Accelerating Compaction on GPUs

The sequential implementation of the *compaction* operation is straightforward and many programming languages and/or libraries provide such an implementation. However, to implement a parallel version, the most challenging issue is to determine the index positions of the selected elements in the output array. According to [2], a prefix-sum approach works very well for a parallel implementation. This prefix-sum approach works as follows: The input data is partitioned into fixed-sized chunks. As the number of bits in the bit mask is unknown at compile-time, a `popcount` operation is executed on each chunked bit mask to determine the number of set bits. To calculate the offset positions where each chunk starts to write back into the output array, a prefix-sum over the popcounts has to be computed next. The prefix-sum is defined as a sequence of numbers, where each element is generated by adding a number to the last element. Using these offset positions, a parallel write-out to the output array can be performed. Thus, a parallel implementation of the *compaction* primitive consists of three phases: **popcount**, **prefix-sum**, and *write-out*.

Since *compaction* only writes back selected data, low percentages of selected data offer a great opportunity for optimizations. Thus, we present an enhanced optimization for this called *smart partitioning for GPU compaction (SPACE)*. *SPACE* is based on the general *prefix-sum* approach with selective optimizations to skip the write-out part of chunks where no elements are selected. These optimizations are beneficial, because popcount and prefix-sum only perform operations on the bit mask and generate intermediate data, while the write-out needs to read the data from the large input array and writes back the final results. In the following, we describe the different variants for each phase for a GPU implementation.

3.1 Popcount

To calculate a final memory offset on a chunk, it is required to know how many data elements occur before a chunk. This is achieved by performing a popcount on the bit mask. A hardware intrinsic function $int\_\_popc(unsigned\ int\ x)$[6] counts the bits set to one in a 32 bit wide element. A popcount kernel across all chunks is the first operation in *SPACE*.

3.2 Prefix-sum

Calculating the prefix-sum requires the popcount intermediate data. The prefix-sum consists of all bits that occurred before the specific data element. It is the

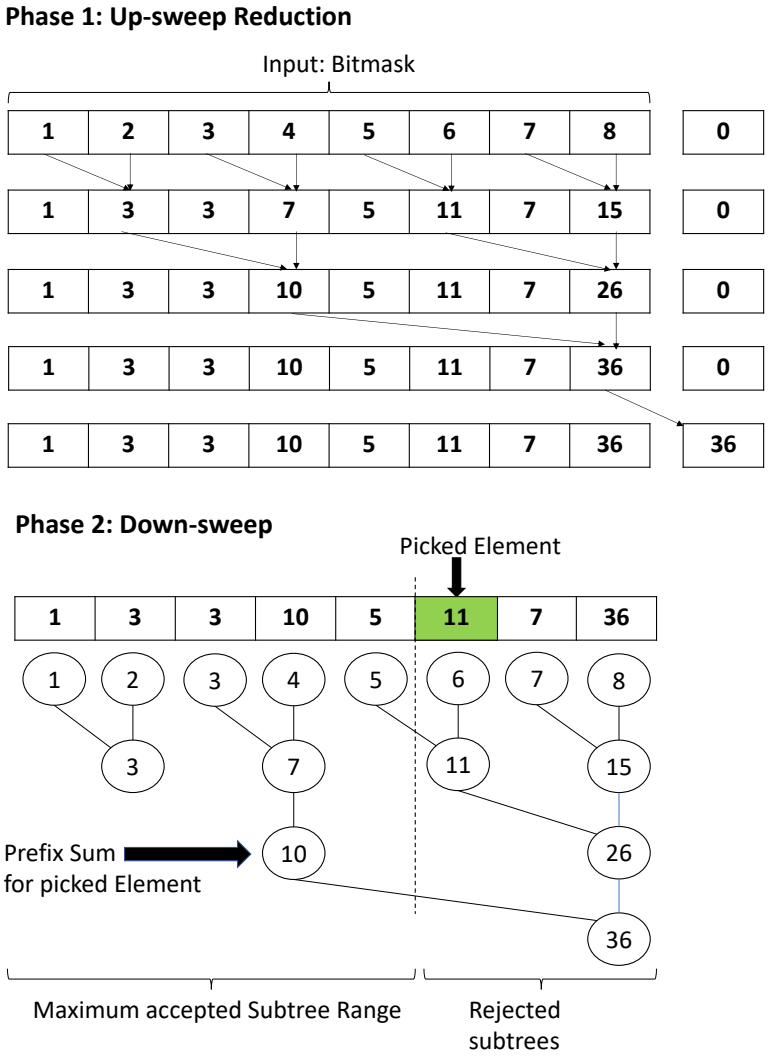

Fig. 3. Two phase prefix-sum is performed. First, up-sweep reduction calculates a partial prefix sum. A single chunk down-sweep calculation is performed for the highlighted element (black arrow, 11). Below the partial prefix sum buffer is the intermediate tree from the up-sweep phase. Sub-trees are either rejected or accepted if they fit. The dotted line indicates the maximum range of fitting sub trees. Thus the red sub-tree with value 11 rejected.

offset in the output array, where each element needs to start to write back data. *SPACE* offers three variants to calculate a prefix-sum.

CUB Prefix-sum: In this variant, the prefix sum is calculated by `CUB::DeviceScan::ExclusiveSum`[4], a function that is provided by the Nvidia SDK.

Two phase prefix-sum: The exclusive prefix-sum starts at 0 for the first entry. First a pyramid reduction is performed as shown in Figure 3. The kernel is called up-sweep reduction. For each layer seen in phase 1 of Figure 3, the kernel is launched once. The resulting tree is an intermediate product that is needed as input for the second phase of the prefix sum calculation. Down-sweep is the second phase that distributes the values of the up-sweep tree through the inner nodes to the leaves of the reduction tree as seen in Figure 3. This creates the final exclusive prefix sum over all data entries. So the final memory offset is the addition of the exclusive prefix-sum entry for data element plus the memory offset of all previous chunks calculated by popcount. An example can be seen in Figure 3. The algorithm attempts to fit the highest amount of largest left sided sub-trees into the index of the chunk to compute. The end result of the *two phase prefix-sum* is an exclusive prefix sum buffer per chunk that can be used to write back all selected elements into the output array. There is also a slightly different version that supports non power of two input sizes.

Partial prefix-sum: Only the up-sweep reduction is performed at this stage, resulting in a partial prefix-sum. If needed, the exclusive prefix-sum is calculated on-the-fly for each element in the write-out phase.

3.3 Write-out

Write-out takes a prefix-sum as input and writes back the final contiguous output array of selected values. Several approaches were implemented for write-out. The key question for write-out is how to efficiently distribute chunks to blocks and threads. Efficient memory access on an NVIDIA GPU requires threads to access memory coalesced. NVIDIA groups 32 threads into a unit called warp. Each instruction is performed on a warp level with 32 threads at a time. If the memory access pattern does not allow warp parallelism, the worst case is a warp level instruction per single element memory access. Thus 32 instructions instead of one. The basic approach is to assign several chunks to each thread. This leads to a bad memory access pattern, as each thread jumps between indices based on the prefix-sum. A more sophisticated approach assigns chunks to warps instead of threads. While thread level synchronization is expensive, warp level synchronization is not. As a chunk can consist of only not selected elements, the write-out can be skipped for that chunk. This can be checked by performing a popcount. As skipping a chunk only needs to read the bit mask and skips reading the larger data array, the amount of memory access can be immensely reduced.

3.4 Overview of SPACE Variants

Variant 1 (base variant approach): A two phase prefix sum and naive write-out kernel are used. Threads are assigned to one chunk. Each thread calculates one element. No chunk is skipped.

Variant 2 (base variant with skipping): Variant 1 is modified by adding skipping of chunks of not selected elements. A popcount kernel that returns 0 bits on a bitmask of an empty chunk indicates which chunks can be skipped.

Variant 3 (asynchronous streaming): The kernel executions are changed to allow usage of the CUDA asynchronous streaming API. Distributing kernels across different CUDA streams allow for kernel level parallelism. However there can be no data dependencies between kernels. Multiple asynchronous CUDA streams are deployed for each kernel that allows concurrent computation.

Variant 4 (optimized read without skipping): Memory access pattern is optimized by using grid striding. Instead of having a thread read additional elements adjacent to each other in memory, the memory access offset per thread is the total number of threads. Adjacent threads attempt to access adjacent data in memory. No calculations are skipped. CUB is used to calculate the exclusive prefix sum.

Variant 5 (optimized read with skipping and partial prefix sum): Based on Variant 4, skipping is added and CUB prefix sum is replaced by the partial prefix sum algorithm. Only the up-sweep reduction is performed before the write-out kernel and the exclusive prefix-sum is calculated on-the-fly in the write out phase.

Variant 6 (optimized read with skipping and two phase prefix sum): Like Variant 5, but the prefix sum is calculated by the two phase prefix sum algorithm. This leads to a fully calculated exclusive prefix sum across all elements before the write-out phase.

Variant 7 (optimized read with skipping and CUB based prefix sum): Like Variant 6, but the prefix sum is calculated by CUB. CUB is used to calculate the complete exclusive prefix sum across all elements.

Variant 8 (optimized read with skipping, optimized write-out and CUB prefix sum): Like Variant 7 with the addition of an alternative version of the write-out kernel. Write-out kernel aims to write-out at least a warp at once.

4 Evaluation

We evaluate our approach on three different data distributions for the bit mask as discussed in 2. Moreover, the selected bits of a bit mask determine how many percent of the total bits are set to one. All approaches allow adjustable percentages of selected data and are benchmarked across a larger number of different percentages of selected data.

4.1 Implementation

All *SPACE* variants are implemented in C++ 17 with CUDA 11 and publicly available [14]. CMake is used to build *SPACE* and NVIDIA nvcc is used as compiler. nvcc calls g++ to compile host code. See Table 1 for detailed information about the used compilers and tools across all platforms. CUDA version 10.x and lower are not able to compile *SPACE*, due to the lack of C++17 support.

GPU	GPU Generation	CMake	CUDA	NVCC	G++
A100	Ampere	3.23.0-rc1	11.2	11.2.67	9.3.0-17
RTX 8000 Quadro	Turing	3.22.2	11.5	11.5.119	9.3.0-17
1070 TI	Pascal	3.22.2	11.5	11.5.50	9.3.0-17
3080	Ampere	3.22.2	11.6	11.6.55	9.3.0-17

Table 1. An overview of evaluated hardware platforms. Version of used build and compilation tools are listed as used in the evaluation.

4.2 Experimental Setup

Data distributions Three different bit mask distributions are investigated: 1 Cluster, multiple Cluster, uniform. 1 Cluster consists of a single cluster of bits set to 1, while the rest of the bit mask remains zero, thus not selected for output. Multiple cluster distributes several clusters equidistant across the bit mask. The number of clusters varies between 2 and 32 clusters. Uniform is based on a uniform distribution that distributes bits across the whole bit mask. All distributions support any data type that is CUDA compatible and between 1 Byte and 8 Byte large. See Table 2 for an overview of which data types we tested. While the size of the input data array is always set to 1 GiB, this results in different sizes for the bit mask. The size of the bit mask is calculated by dividing 1 GiB by the size of one data element. For example a 1 GiB Array of uint64_t results in 16 MiB bit mask. Elements from the data input array are randomly generated.

Hardware platforms We use different hardware platforms with CUDA 11 as shown in Table 1.

Data type	Size of Element [Byte]	Total bit mask size [MiB]
u8	1	128
u16	2	64
u32	4	32
u64	8	16
Integer	4	32
Float	4	32
Double	8	16

Table 2. Size of evaluated data types and corresponding bit masks.

4.3 Experimental Methodology

As one *SPACE* algorithm consists of several kernel calls, we measure the total runtime of all kernel calls with CUDAevents. The throughput is calculated as

$$throughput = \frac{datasize(bit\_mask[GiB] + input\_data[GiB])}{total\_runtime\_of\_one\_algorithm[s]}$$

This calculation does not account for skipping chunks. As a result, the calculated throughput can exceed the maximum memory bandwidth. The goal of this throughput calculation is to achieve a good comparability of all measured SPACE algorithms and CUB. All data is present on GPU. Data generation is not part of the measurement. Additionally, the input data is partitioned into chunks varying between 512 and 4096 elements.

4.4 Results

In Figure 4 an overview of the evaluation of all *SPACE* variants is shown. A grid of graphs is shown with different GPUs and data distributions. Percentages of selected data are within 1% and 97% with increments of 4. Variants 1 and 2 overall perform very poorly across all devices. Variant 3 with async CUDA streams outperforms Variants 1 and 2 but is often slower than CUB for higher percentages of selected data. Variants 4,5,6,7 mostly outperform CUB even for very high percentages of selected data, variant 8 is the best performing *SPACE* variant and outperforms all other SPACE variants. Even for higher percentages of selected data Variant 8 outperforms CUB in all cases on the A100 GPU. In case of the Quadro RTX 8000 GPU, CUB is ahead if the percentage of selected data exceeds 71% in a clustering approach, or 21% for uniform distributions. For the 1070 ti, CUB is ahead if the percentage of selected data exceeds 53% in uniform and never for clustering distributions. In case of the 3080 GPU, CUB is slower if percentage of selected data is 81% and smaller for a single cluster. The break even point for uniform is at 33% for a 3080 GPU. The ideal improvement for 1% selected data and single cluster distribution is 12.01x in favor of SPACE variant 8. In case of uniform distribution the improvement for 1% is 3,19x in favor of SPACE variant 8.

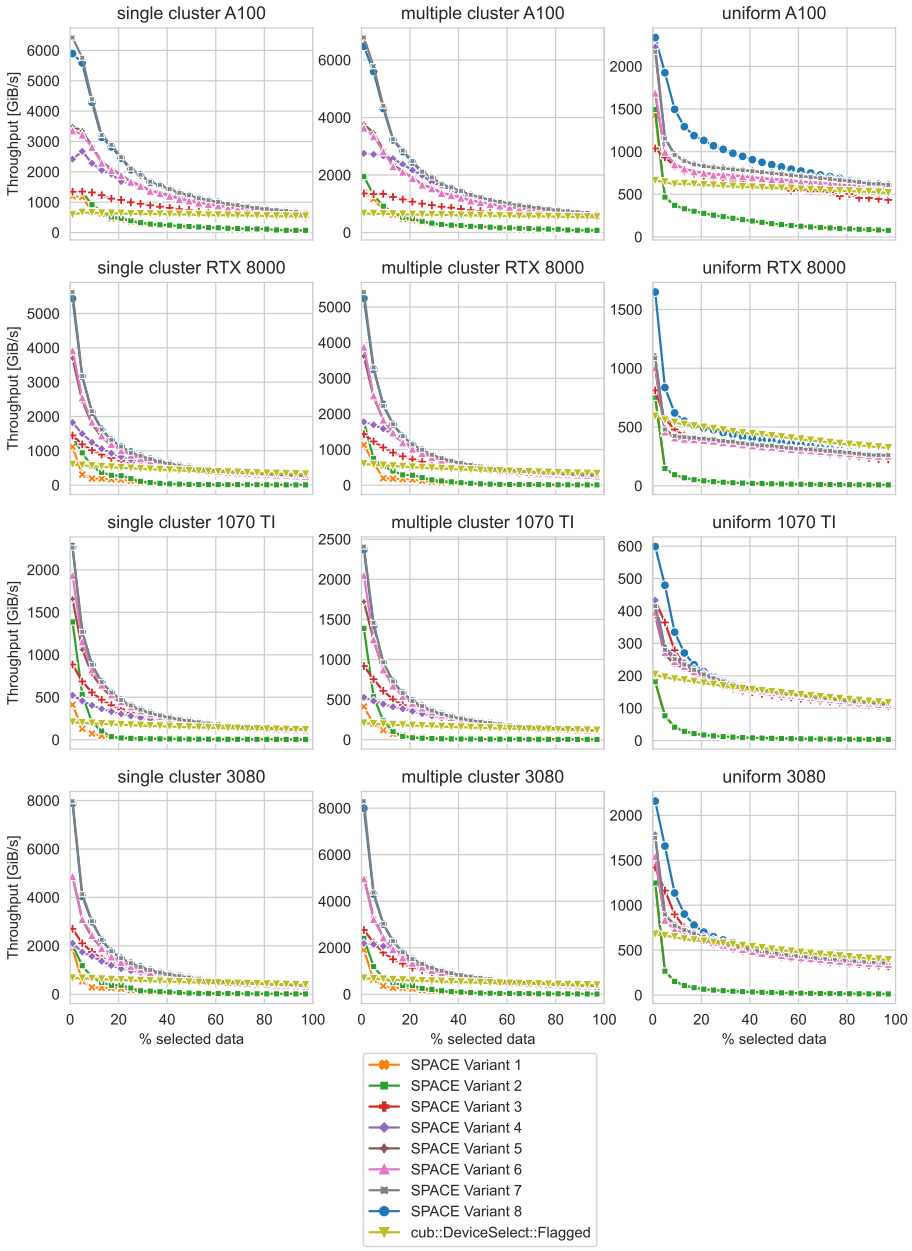

Fig. 4. Overview of all SPACE algorithms for all hardware platforms and data distributions. Datatype is unsigned integer 32 bit.

Very low percentages of selected data Figure 5 shows an additional experiment with very low percentages of selected data. All experiments were conducted on a RTX 8000 GPU with 32 bit unsigned integers. Reducing the percentage of selected data further from 1% to 0.1% more than doubles the throughput for both clustering approaches. Uniform reaches above 5000 [GiB/s] only at 10^{-3}%, while both clustering approaches reach it at 0.1%. The highest improvement is achieved for the second lowest percentage of selected data with 10.40x improvement for uniform. CUB achieves 603 [GiB/s] at 10^{-5}% while SPACE 8 reaches 6272 [GiB/s] throughput. In case of single cluster distribution CUB achieves 553 [GiB/s] at 10^{-5}% while SPACE 7 reaches 6647 [GiB/s]. This leads to an improvement of 12.02x in this extremely favorable case. We conclude that decreasing selected data yields diminishing returns. For 10^{-4}% and lower percentages of selected data there is not much improvement to gain. For uniform and single cluster, 10^{-6}% was slightly slower than 10^{-5}%. CUB performs with similar performance across different distributions and percentage of selected data.

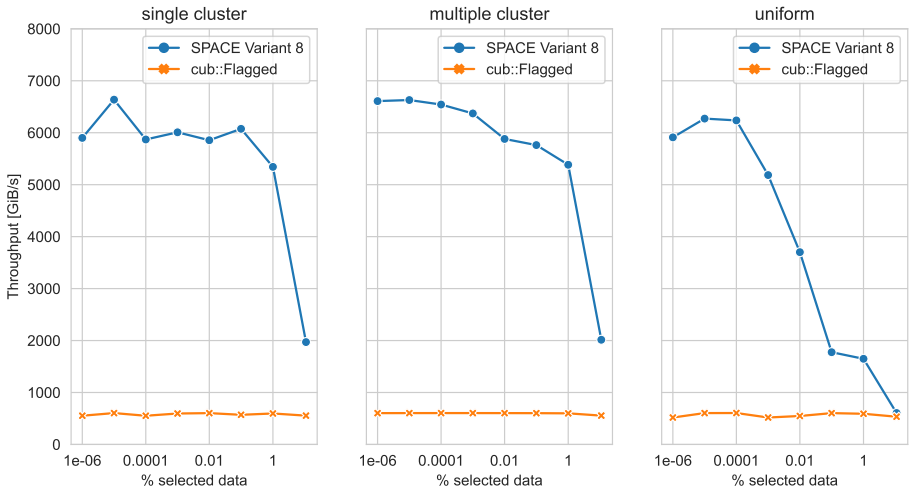

Fig. 5. Very low percentages of selected data in logarithmic scale on x-axis. Measured on RTX 8000. Datatype is unsigned integer 32 bit.

Influence of data types As shown in Table 2, we evaluated our algorithm across 7 different data types. Figure 6 shows the performance of SPACE 8 against CUB for different data types. Skipping is beneficial across all data types. Peak performance for 1% selected items greatly varies between data types. Double achieves 8860 [GiB/s] at 1%, float achieves 5897 [GiB/s] and uint8_t reaches 3313 [GiB/s]. The data type has massive influence on the overall performance. Skipping is beneficial and results in significant improvement compared to CUB across all data types.

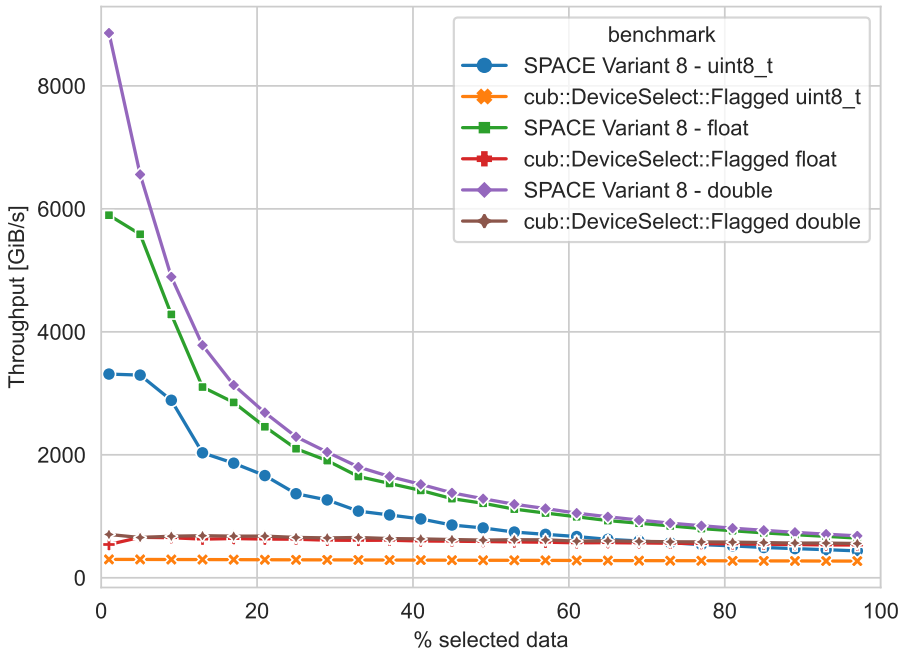

Fig. 6. Comparision of SPACE 8 against CUB for uint8_t, float and double data types. Single Cluster distribution is used on A100 GPU.

Influence of chunk sizes SPACE variants calculate skipping based on chunk size, which determines how many elements are used for each warp. The chunk sized has been varied between $[512, 1024, 2048, 4096]$ elements in our evaluations. A data set of 32 bit unsigned integers with single cluster distribution on A100 is picked to evaluate the influence of chunk sizes. Over all percentages of selected data and all kernels the best chunk size has an average improvement of 12%. However the largest differences are measured at low percentages of selected data and poor performing variants like SPACE 1-3. For the best performing two variants SPACE 7 and SPACE 8 the average improvement from worst to best chunk size is 4.5%. If percentages of selected data below 25% are excluded the average worst to best improvement is reduced to 1.4%. For the best performing variants SPACE 7 and SPACE 8 a chunk size of 512 elements is the fastest choice. SPACE 1 und SPACE 2 achieve the best performance with a chunk size of 4096 elements.

Influence of CUDA configurations CUDA Configurations play a signficant role for the overall runtime. Each variant consists of a number of different kernel calls. All kernel calls were measured for a large variety of different configurations. The best configurations per kernel were used for our evaluations.

Selection of datasets Real world data sets are not included in our experiments. We have shown that the following input characteristics determine the performance of SPACE: data types, bitmask data distribution and percentage of selected data. As the experiments range from worst case to best case for all three characteristics, a real world data set would fit within the current spectrum of parameters. As a consequence we did not include real world data.

5 Related Work

Bakunas et al. classify compaction on GPU into two categories. [2] [1]. Atomic based approaches that have a global memory offset index, which all writings threads increment in parallel using atomics, and prefix sum based approaches. On the evaluated NVIDIA Kepler platform, atomics with a single global counter create a major bottleneck. By leveraging warp level parallelism and changing to a prefix-sum based approach, improved performance was achieved compared to global counter atomics. THRUST, an open source compute library is used as part of the algorithm. Merril et al. introduce CUB select if, which computes a compaction based on a selection with a functor to a compare predicate, instead of a bit mask [12]. Their approach outperforms THRUST significantly. The common denominator for all these approaches is, that they do not optimize for low amounts of selected data unlike SPACE.

6 Conclusion and Summary

The goal of SPACE is to offer a specialization that accelerates compaction on GPU for low percentages of selected data. Skipping memory access to the input data array by analyzing the bit mask is the key idea to gain performance. Great improvements for low amounts of selected data are achieved. In case of 1% the improvement is up to 10.05x. For very low % of selected data the improvement is 12.02x on a NVIDIA Quadro RTX 8000. Depending on the GPU device, our approach is slightly faster than CUB even for data sets that do not benefit from skipping. Source code for all kernels, scripts for experiments, scripts for visualizations and all raw data is provided on our github [14].

Acknowledgements and Data Availability Statement

The datasets and code generated during and/or analyzed during the current study are available in the Figshare repository:
https://doi.org/10.6084/m9.figshare.19945469 [7].
This work is funded by the German Research Foundation within the RTG 1907 (RoSI) as well as by the European Union's Horizon 2020 research and innovative program under grant agreement number 957407 (DAPHNE project).

References

1. Bakunas-Milanowski, D., Rego, V., Sang, J., Chansu, Y.: Efficient algorithms for stream compaction on gpus. International Journal of Networking and Computing **7**(2), 208–226 (2017)
2. Bakunas-Milanowski, D., Rego, V., Sang, J., Yu, C.: A fast parallel selection algorithm on gpus. In: 2015 International Conference on Computational Science and Computational Intelligence (CSCI). pp. 609–614. IEEE (2015)
3. Choi, K., Yang, H.: A GPU architecture aware fine-grain pruning technique for deep neural networks. In: Euro-Par Conference. Lecture Notes in Computer Science, vol. 12820, pp. 217–231 (2021)
4. CUB: cub::DeviceScan::ExclusiveSum documentation: `https://nvlabs.github.io/cub/structcub_1_1_device_scan.html\#a02b2d2e98f89f80813460f6a6ea1692b`
5. CUB: Main Page: `https://nvlabs.github.io/cub/index.html`
6. CUB: Main Page: `https://docs.nvidia.com/cuda/cuda-math-api/group__CUDA__MATH__INTRINSIC__INT.html`
7. Fett, J., Kober, U., Schwarz, C., Habich, D., Lehner, W.: Artifact and instructions to generate experimental results for the euro-par 2022 paper: "accelerating parallel operation for compacting selected elements on gpus". In: European Conference on Parallel Processing. Springer (2022), `http://doi.org/10.6084/m9.figshare.19945469`
8. Guo, W., Li, Y., Sha, M., He, B., Xiao, X., Tan, K.: Gpu-accelerated subgraph enumeration on partitioned graphs. In: SIGMOD Conference. pp. 1067–1082 (2020)
9. Hertzschuch, A., Hartmann, C., Habich, D., Lehner, W.: Simplicity done right for join ordering. In: 11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings (2021)
10. Hu, L., Zou, L., Liu, Y.: Accelerating triangle counting on GPU. In: SIGMOD Conference. pp. 736–748 (2021)
11. Lo, S., Lee, C., Chung, I., Chung, Y.: Optimizing pairwise box intersection checking on gpus for large-scale simulations. ACM Trans. Model. Comput. Simul. **23**(3), 19:1–19:22 (2013)
12. Merrill, D., Garland, M.: Single-pass parallel prefix scan with decoupled look-back. NVIDIA, Tech. Rep. NVR-2016-002 (2016)
13. Sistla, M.A., Nandivada, V.K.: Graph coloring using gpus. In: Euro-Par Conference. Lecture Notes in Computer Science, vol. 11725, pp. 377–390 (2019)
14. SPACE Github: `https://github.com/yogi-tud/SPACE/`
15. Turing Tuning Guide: CUDA Toolkit documentation: `https://docs.nvidia.com/cuda/turing-tuning-guide/index.html`
16. Ungethüm, A., Pietrzyk, J., Damme, P., Krause, A., Habich, D., Lehner, W., Focht, E.: Hardware-oblivious SIMD parallelism for in-memory column-stores. In: 10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings. www.cidrdb.org (2020)
17. Zhou, K., Hou, Q., Wang, R., Guo, B.: Real-time kd-tree construction on graphics hardware. ACM Trans. Graph. **27**(5), 126 (2008)