Accelerating GPU Data Processing using FastLanes Compression

Azim Afroozeh
azim@cwi.nl
CWI
Amsterdam, Netherlands

Lotte Felius
felius@cwi.nl
CWI
Amsterdam, Netherlands

Peter Boncz
boncz@cwi.nl
CWI
Amsterdam, Netherlands

ABSTRACT

We show that compression can be a win-win for GPU data processing: it not only allows to store more data in GPU global memory, but can also *accelerate* data processing. We show that the complete redesign of compressed columnar storage in FastLanes, with its fully data-parallel bit-packing and encodings, also benefits GPU hardware. We micro-benchmark the performance of FastLanes on two GPU architectures (Nvidia T4 and V100) and integrate FastLanes in the Crystal GPU query processing prototype. Our experiments show that FastLanes decompression significantly outperforms previous decompression methods in micro-benchmarks, and can make end-to-end SSB queries up to twice faster compared to uncompressed query processing – in contrast to previous work where GPU decompression caused execution to slow down. We further discovered that an access granularity of decoding vectors of 1024 values is too large for a single GPU warp due to register pressure. We mitigate this here using mini-vectors – a future work question is how to further reduce this granularity with minimal impact on efficiency.

ACM Reference Format:
Azim Afroozeh, Lotte Felius, and Peter Boncz. 2024. Accelerating GPU Data Processing using FastLanes Compression. In *20th International Workshop on Data Management on New Hardware (DaMoN '24), June 10, 2024, Santiago, AA, Chile.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3662010.3663450

1 INTRODUCTION

The FastLanes project[1] is working towards a new analytical data format to better suit modern workloads and modern hardware. In the first paper on FastLanes, that describes its novel and completely data-parallel columnar encodings, we showed its very high performance using data-parallel SIMD instructions on CPUs [1].

In this paper, we evaluate and optimize FastLanes decompression on GPUs. In the past decades there has been a lot of research on database processing on GPUs and also numerous start-ups; but database workloads have not migrated to GPUs yet; instead, a flood of ML workloads have propelled GPUs to center stage in data centers. These workloads consume a steady stream of data in ML training and inferencing, producing a growing call for data formats that are compatible and performant on both CPUs and GPUs [11].

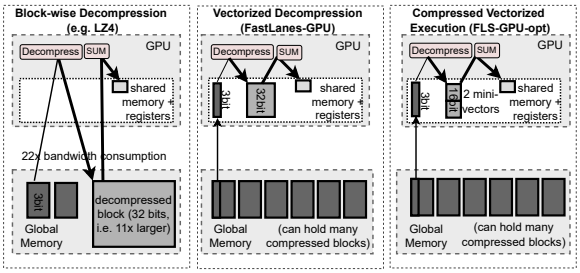

Figure 1: Three different ways of decompressing data on the GPU. Left shows that decompressing into global memory (as typically done in GPU decompression) can cause very high memory bandwidth consumption. The middle shows how vectorized decompression avoids spilling back to global memory, by directly processing the decoded data. The rightmost shows how the pressure on GPU registers and shared memory (cache) can be reduced, by (i) reducing the decoding batch size, and (ii) by decoding into thin (<32-bit) data types.

Data formats that are now ubiquitous in data lakes, such as Parquet and ORC, were originally designed for use on CPUs [11]. While these formats harbor multiple good ideas (schemas, statistics, columnar storage, compression, vectorized decoding) they have limitations that hurt GPUs; particularly the fact that their column encodings do not compress data enough, and therefore their datapages are further compressed with general-purpose block-based schemes (gzip, zstd, lz4 or snappy), which are GPU-unfriendly.

Data compression is an attractive proposition for GPUs: they typically have smaller RAM ("global memory") than the host CPU machine, such that storing compressed data alleviates a capacity bottleneck. Further, data is moved into the GPU over the PCIe bus, so having to move less data thanks to compression helps to reduce that bottleneck. But this hinges on the capability of the GPU to efficiently decompress the data, ideally incrementally, when it is processed. As the left picture in Figure 1 shows, however, block-based compression operates on a coarse granularity that is too large to fit into the shared memory (the on-die GPU cache), whose size is typically just tens of KB, shared among 32 threads. Note that e.g. parquet-mr typically creates pages of 1MB, that get (de)compressed as one block, which exceed this size. This means that the uncompressed result of decompression must spill back to GPU global memory, significantly increasing the memory bandwidth usage inside the GPU, because the decompressed data is much larger than the compressed data. In the depicted example, when a compressed column that takes 3 bits per value, is decompressed into the standard 32-bit integer that GPUs manipulate; this will transfer in total 22x the compressed bandwidth (3+32bits for decompressing, plus

[1] github.com/cwida/FastLanes

DaMoN '24, June 10, 2024, Santiago, AA, Chile

ACM ISBN 979-8-4007-0667-7/24/06
https://doi.org/10.1145/3662010.3663450

32bits upon use). There are three problems here: (i) decompression algorithms like lz4 are essentially sequential (have many control- and data-dependencies) and therefore run inefficiently on GPUs (ii) materializing buffers of uncompressed data in global memory as depicted, wastes scarce global memory capacity. (iii) transferring uncompressed data for processing (here SUM) into GPU kernels, can make these bandwidth-bound.

In FastLanes we pursue cascading (recursive) application of column encodings (FOR, DELTA, RLE, DICT) to remove the need for general-purpose compression like lz4 for getting good compression ratios [4]. FastLanes encodings are fully data-parallel, even eliminating the sequential dependencies that are normally present in DELTA decoding and RLE. This data-parallelism works great on CPU SIMD instructions but conceptually also fit the SIMT GPU model, where 32 threads that make up a warp and execute the same instructions in lockstep; without any data-dependencies between the threads. FastLanes stores data in 1024-value vectors, where 32 or more adjacent values can be decompressed completely independently of each other. We propose the integration of FastLanes decoding in GPU data processing as depicted in the middle of Figure 1 using *vectorized decompression*: decompression as the first step of data processing, where one vector of data is decompressed into registers or shared memory and is directly consumed from there, without spilling back to global memory.

In this paper, we show that FLS-GPU decompressing a vector of 1024 values can easily be too coarse-grained for GPUs. If each of the 32 threads in a warp decodes 32 values, these should be stored in GPU registers or shared memory, in order not to spill into GPU global memory. However, depending on the GPU architecture, this can already be close to the average available registers per thread (see last row of Table 3). Worse, the consuming data processing task (like a database query, or ML inferencing) typically needs *multiple* columns, which increases memory pressure on registers and shared memory – which may also hold e.g. lookup-tables. This then causes GPU register spilling, leading to increased memory latency and/or a reduction of scheduled tasks, leading to GPU under-utilization. Therefore, we developed GPU-specific optimizations (FLS-GPU-opt) that reduce register and shared memory pressure. Key ideas are: (i) dividing a vector into mini-vectors and decompressing mini-vector at-a-time; (ii) thinking beyond the standard 32-bits GPU data type and simulating smaller data types; thus considering 16- and 8-bits data widths. We (iii) also increased the block-width from 32 to 128 or even 256 in order to reduce scheduling overhead.

Our main **contributions** in this paper are:

- **Generating and Micro-benchmarking FLS-GPU code.** We use a code generator to generate the C++ FastLanes CPU encoding and decoding methods for all relevant bit-widths statically at compile-time. This code generator was extended such that it can generate CUDA code. We show that FLS-GPU outperforms the current state-of-the-art GPU decompression algorithms GPU-FOR and GPU-DFOR by performing micro-benchmarks where we decompress into *global memory*, *shared memory* and GPU registers.
- **Integrating and Optimizing FastLanes in Crystal.** Further, FastLanes bit-unpacking is integrated into Crystal [8]. To increase performance we tested *compressed execution*, partitioning 1024-tuple vectors into *mini-vectors*, increasing the block size, and the sorting of columns to simulate RLE and obtain a better compression ratio. These optimizations enable to release pressure from registers and *shared memory*, and are ultimately combined with optimizations proposed by Crystal-opt [2].

Outline. First, we discuss Crystal, Crystal-opt and Tile-based decompression: the fastest state-of-the-art GPU academic database systems and decompression scheme respectively. We also summarize the FastLanes layout that uses *interleaved* bit-packing and the *transposed layout* to eliminate dependencies from the DELTA and RLE encodings. Then, we discuss how we adapted FastLanes to be compatible with GPUs and we propose optimizations to improve the performance of FastLanes on Crystal. Lastly, we discuss the obtained results and share our findings and ideas for future work to further optimize FastLanes for GPU-based data processing.

2 BACKGROUND

In this section, we shortly explain the GPU memory hierarchy and basic principles of `CUDA`. In addition, the lightweight compression (LWC) algorithms used by FastLanes are briefly explained, along with an explanation of the intrinsics of FastLanes en/decoding.

2.1 GPU Programming

In this paper we use CUDA for programming the GPU [5]. CUDA only works for NVIDIA hardware, so in an increasing heterogeneous hardware landscape, a more portable API such as Vulkan [10] could be considered. However, CUDA offers a more mature toolchain and higher-level programming model; hence we use it for this initial study of FastLanes on GPUs.

CUDA splits the written code in two parts: the *device code*, used to program the GPUs itself, and the *host code*, which is CPU code. The host code takes care of initialization and launching of the kernels in the program and allocating memory regions. The device code is written as a sequential program, thus for a single thread, but executed for multiple threads at once, a model known as Single-Instruction-Multiple-Threads (`SIMT`). CUDA *virtualizes* physical hardware. A *thread block* in CUDA is a virtualized streaming multiprocessor (SM). It is typically recommended to use at least 128 threads in a block, to limit scheduling overhead.

Each SM contains cores, a register file, a warp scheduler, data caches, instruction buffers and texture units. A SM can therefore be considered a whole machine on itself. Thread blocks are launched on a single SM and are independent of each other, meaning that they run to completion without preemption. A *thread* itself is a virtualized scalar processor which contains its own registers. Threads are grouped into *warps*, which is the basic unit of execution for a single `SM`. A warp is a unit that consists of 32 threads that all run concurrently on a SM. All threads in a warp execute the same instruction when running a kernel. If the kernel contains branches (if-then-else), *thread divergence* can occur if some threads take the if- and other the else-branch. The threads must execute all instructions in lock-step, but the instructions off the chosen paths become no-ops. This also affects while-loops: all threads execute as long as the longest loop in the warp.

The memory hierarchy of a GPU differs significantly from the CPU memory hierarchy. In the memory hierarchy of the GPU, each thread contains its own 32-bit registers. Next to the amount of registers per thread, each thread contains its own *local memory* (`lmem`). `Lmem` is not really a memory – its bytes are stored in the GPU main memory (global memory). The name local memory refers to the memory where registers and other data from a thread is spilled, when e.g. a thread exceeds the register limit. The main differences between `lmem` and global memory however are (1) stores are always cached in L1 cache and (2) addressing is resolved by the compiler itself. If the L1 cache is full, a line gets evicted to L2 cache or DRAM. In this case, a store incurs multiple writes.

Each thread block contains its own *shared memory* which functions as a programmable L1 cache of usually a few tens of KB. Shared memory thus stores data which is accessible for *all* threads within a thread block and gets wiped when a new block is executed.

Unlike the L1 cache, the L2 cache is shared among all SMs. There is *global memory*, which is accessible to *any* thread at all times. Global memory is the main memory where data is stored when loaded from the CPU into the GPU. The bandwidth to the GPU is typically a few factors higher than main memory attached to a CPU (often using HBM technology). Data is transferred using wide cache-lines (typically 128 bytes), and best access is achieved if the threads in a warp load adjacent data (*coalesced* memory access).

Transferring data, and communication between host (CPU) and device (GPU) in general, happens via the PCI/e bus. However, such I/O should be minimized where possible as the PCI/e bandwidth is much lower than global memory bandwidth.

2.2 Lightweight Compression using FastLanes

This section explains state-of-the art approaches of different lightweight compression schemes on the GPU. In addition, it explains how FastLanes internally works to provide efficient en/de-codings for these lightweight schemes.

Common Lightweight Compression Methods. Analytical database systems make extensive use of compression to reduce memory footprint when storing and accessing data. However, general-purpose compression methods such as Snappy or LZ4 are computationally expensive for decompression at runtime. Therefore, lightweight compression methods or *encodings* such as Bit-packing, DICT, DELTA and Run-Length Encoding (RLE) are typically used as a first step. Exploiting the fact that the the actual domain of data stored close together is often much smaller than what their data type can represent, Bit-packing allows to represent larger data types in fewer bits. It is typically the lowest-level encoding applied to stored data. The other encodings work on top of bit-packing. DICT uses a dictionary, holding unique values, and represents the original data as (bit-packed) integer codes, which are positions in this dictionary. DELTA encoding exploits value locality, by only storing the (bit-packed) difference between subsequent values. Note that during DELTA decoding subsequent values are dependent on its predecessor (a data dependency). RLE encoding, finally, is particularly effective on sorted data with lots of repeating values. It encodes the repeating values in so-called `run-lengths`. Note that decoding RLE requires a loop over the run-length, which on GPUs can lead to thread divergence .

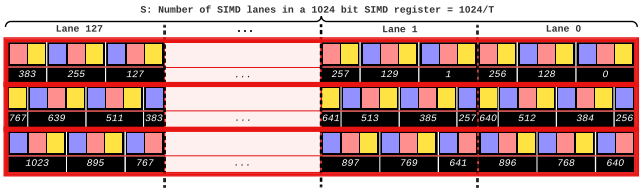

Figure 2: FastLanes bit-packs a vector by interleaving its values (vector positions in black below) over many lanes; in this example 128 lanes of 8-bits [1]. On GPUs this leads to all threads in a warp accessing directly adjacent lane data, which is the optimal memory access pattern.

FastLanes. FastLanes is an open-source library that provides lightweight encodings, in a fully *data-parallel manner*. Even though its CPU source code is scalar, this property allows compilers to *auto-vectorize* it into SIMD instructions, such as AVX512 on AMD and Intel, and NEON or SVE on ARM. SIMD instruction on CPUs can execute an operation on multiple data items, stored adjacently in 8-, 16, 32- or 64-bit *lanes*, in one instruction.

The first step in decoding is *un-packing* densely packed bits into 8-, 16-, 32- or 64-bits (byte-addressable) integers. Standard bit-packing schemes, however, pack bits of adjacent values tightly together right after each other. In principle, SIMD instructions can perfectly support bit (un)packing as they support the required operations (AND, OR, SHIFT). The problem here is that to produce the original value sequence with SIMD instructions, adjacent values will reside in the same SIMD lane. To avoid this, *interleaved* bit-packing distributes subsequent values round-robin over subsequent lanes. Thus, a data-parallel unpacking kernel can produce subsequent values from subsequent lanes, without needing (expensive) inter-lane data transfers.

After values have been bit-unpacked; they can be decoded using lightweight schemes like DICT, RLE and DELTA. The latter two schemes have proven challenging for SIMD instruction sets, because of the sequential dependencies in DELTA and the loops required to decode RLE (SIMD does not support loops). For example, if we encode a column of [1, 2, 3, 4, 5] into [1, 1, 1, 1, 1] using DELTA, we can only restore the original values if we know the predecessors of the value that we want to decode. To tackle this issue, FastLanes proposes the *Unified Transposed Layout* that removes the data dependency by using a special permuted order for the 1024 tuples in a vector (Figure 3 shows a simplified permuted order for 16 tuples). Thanks to this, every lane produces independent running sums to perform DELTA decoding. FastLanes further maps RLE coding to DELTA coding, to also make RLE fully data-parallel [1].

By removing all data dependencies, FastLanes achieves ultra-high performance [1]: up to 60 values per CPU cycle single-threaded. We think this is important, because in the vectorized decompression model, most of the computational time should go in the data processing (i.e. database query processing, or machine learning), and only a minority in decompressing, if we want to alleviate a memory bottleneck and make data processing faster. Ultra-fast decoding also facilitates our move towards *cascading* column encodings; that implies multiple decoding kernels are invoked per column – this otherwise could become expensive. As an example of cascading encoding, the thin integer codes arrays used in DICT, as well as its dictionary array can be further encoded, e.g. using RLE or DELTA.

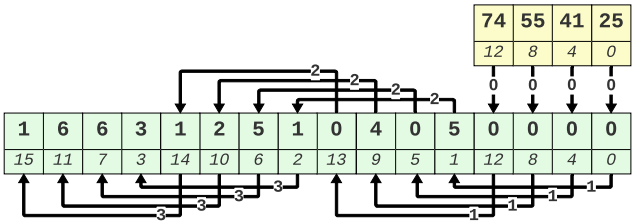

Figure 3: Transposed Layout of a 16-value vector for making DELTA decoding data-parallel[1]. Starting with a base stored in a header (yellow), DELTA has to sum up all differences, a sequential task. By storing multiple bases, and reordering the tuples in vector (small below numbers are positions) we can now execute this with four 4-way SIMD additions. The FastLanes layout (that stores 1024 values, not 16) provides enough parallelism to let all GPU threads in a warp do completely independent sums.

The work in BtrBlocks [4], which does not use data-parallel encodings, has established that cascading lightweight encodings can achieve compression ratio's comparable to Parquet with LZ4. In ongoing work on FastLanes, that adopts cascading encoding, we have confirmed this finding.

3 RELATED WORK

There has been only a modest amount of research into compression schemes for GPU-based database systems. *Fang et al.* [3] implement nine compression schemes suitable for the GPU. These schemes are divided between main schemes and auxiliary schemes. These schemes include null-surpression (NS) of fixed and variable length, DICT, bitmap and RLE. Auxiliary schemes include FOR, DELTA, SEP and SCALE. *Shanbhag et al.* [9] propose GPU-FOR, GPU-DFOR, and GPU-RFOR where the latter two do respective DELTA and RLE. None of the previous GPU compression schemes [3, 7, 9] has changed the value order (i.e. interleaving or transposing) to make decompression data-parallel.

Crystal. [8] is the current state-of-the-art academic GPU-based query processing prototype [2]. It consists of hard-coded CUDA kernels implementing each of the SSB benchmark queries [6]. To benefit from the parallelism provided by GPUs, Crystal adopts a *tile-based* execution model. Here, a *Tile* consisting of 128 tuples is the basic a unit of execution and is processed as a whole in a single thread block, which in its turn is partitioned by the GPU into warps. Processing at the granularity of tiles aligns with the 128-byte GPU cache lines and leads to reduced cache misses and *coalesced memory accesses*. Before performing any operations, a tile of 128 values is loaded directly into the available registers of each thread. Loading a tile enables coalesced memory access and loading into registers avoids an extra pass to shared and global memory. Crystal is not a production system, since all queries are hand-written. Other significant limitations are e.g. hard-coded parameters for hash tables and the lack of support for data types other than 32-bits integers.

Crystal-opt. The authors in [2] pointed out that Crystal is memory-bound, and there is room for improvement. Crystal (i)) loads unnecessary data from DRAM if filter predicates are selective and (ii) it bypasses the L1 cache which could be exploited for operating on intermediate results [2]. In Crystal-opt, query performance is improved roughly 2x, making it currently the fastest GPU-based query execution prototype for SSB queries. The first optimization is predicated loads (*PredLoad*, essentially predicate push-down) that avoid loading tuples from global memory that are already disqualified by a predicate. While such an if-then-else test introduces GPU thread divergence, doing so for avoiding unnecessary loads is a good trade-off. A somewhat less important optimization is to check whether an entire tile has no more qualifying tuples. This is only beneficial if *all* threads in a warp can be terminated early (a quite rare phenomenon). A second optimization to disable L1 cache bypass on certain queries (manually), for those queries that profit from this (a manually tuned decision).

Tile-Based Decompression. [9] introduced *tile-based decompression* to reduce memory bandwidth consumption in Crystal data scans. Three new compression schemes are introduced, based on the tile-based execution model that combine bit-packing and FOR: GPU-FOR, GPU-DFOR and GPU-RFOR. The latter two denote DELTA and RLE, respectively. Bit-unpacking in tile-based compression is a generic function that has bit-width as a parameter. For any tuple, it loads two values, computes two shifts and two AND-masks to apply to these values, and OR-s them together; which is a worst-case approach in terms of bit-unpacking. In contrast, a typical CPU-based vectorized bit-unpacking function would be templated by bit-width (rather than receive it as a dynamic parameter) and internally contain a series of hard-coded LOAD-SHIFT-AND-STORE instructions (with occasional OR work, only applied to those bit-sequences that cross a word boundary) to unpack all tuples in a vector – which is computationally much more efficient than computing the shift amounts and the masks for each tuple and always doing an OR. The authors argue that GPU decompression is bandwidth-bound and thus such computational expense is of no importance; an argument that in our view is only correct when compressing back into global memory (leftmost Figure 1, which we recommend not to do). The Tile-Based micro-benchmarks in [9] are decompressing into global memory – in this paper we re-do those micro-benchmarks using the alternatives of decompressing into registers and into shared memory - where the query processor directly consumes it (middle of Figure 1: vectorized decompression). Finally, the end-to-end SSB results reported in [9] are 35% slower than running Crystal on uncompressed data; which is explained as something that should be expected – however this paper shows that SSB queries can get *faster* thanks to compression. This comes in addition to the compression benefit of being able to store 2-4x more data in GPU memory (and a reduction of the PCIe bottleneck when data is moved into the GPU).

4 FASTLANES ON GPU

In this section, we explain our first implementation of FastLanes on the GPU, which is available in our GitHub repository[2]. Moreover, we perform micro-benchmarks to test three different approaches (global-to-global, global-to-shared and global-to-registers) to determine which approach minimizes latency when decompressing data into memory.

[2]https://github.com/cwida/FastLanesGPU

Table 1: *global-to-global* decoding 3 to 32 bits (256M values)

GPU	GPU-FOR	FLS-BP	GPU-DFOR	FLS-DELTA
T4	7.80 ms	5.89 ms	12.36 ms	6.05 ms
V100	1.60 ms	1.60 ms	1.77 ms	1.64 ms

Table 2: *global-to-shared* decoding of 3 to 32 bits (256M values)

GPU	GPU-FOR	FLS-BP	GPU-DFOR	FLS-DELTA
T4	10.02 ms	2.42 ms	14.63 ms	4.16 ms
V100	1.22 ms	0.44 ms	1.90 ms	0.63 ms

Table 3: Specifications for Tesla T4 and Tesla V100 GPUs

Model	T4	V100
Memory	16GB	16GB
Memory Bandwith	320.0 GB/s	900.0 GB/s
L1 size (per SM)	96 KB	128 KB
L2 size	6 MB	6 MB
SM count	40	80
Max. Warps/SM	32	64
Max. Blocks/SM	16	32
Max. Threads/SM	1024	2048
Max. 32-bit reg/SM	65536	65536
Avg. registers/thread	64	32
Max. registers/thread	255	255

4.1 Initial Implementation

FastLanes for CPUs leans into the *Single Instruction Multiple Data* (SIMD) capabilities of CPUs to decode multiple values in one pass. GPUs, however, are based on the SIMT model. To exploit the SIMT parallelism provided by GPUs, we assign a vector of 1024 values to a block. Since we assign one warp per block, every thread in a warp decodes 32 values and thus functions as a so-called *lane* in FastLanes. This means that at least 32 values are decoded in parallel in every step within a warp, when decoded into 32-bit integers. Thanks to use of interleaved bit-packing, this leads to coalesced memory access. Decoding into 1024 32-bits values still fits in GPU registers. We note that GPU systems so far focus on 32-bits data processing, since this is the native data type for GPUs. However, FastLanes has the capability of achieving data-parallelism using scalar instructions, i.e., it can use 32-bits instructions to decode 4x8bit and 2x16bit lanes per GPU thread. This capability can be used in GPUs to decode into thinner data-types than 32-bits, reducing GPU shared memory and register pressure; while at the same time performing 2x or 4x more operations per instruction.

4.2 Micro-benchmarks

We use two different GPUs for our benchmarks: the Tesla T4 and Volta V100, from which the exact specifications are shown in Table 3. We benchmark both FastLanes Bitpacking (FLS-BP) and FastLanes DELTA (FLS-DELTA) using C++20, nvcc and CUDA 12.3 for the implementation.

Measured execution time. In the micro-benchmarks we measure the execution from the moment the data is loaded into global Memory of the SM. Loading data from and to the CPU is thus not taken into account. Therefore, the PCI/e bandwidth is not a bottleneck in any of the experiments. The measured time is thus from the time the kernel is executed, until the execution is finished and the final results are written back to global memory. To measure the execution time per kernel we make use of the Nvidia Nsight Compute CLI (`ncu`).

Global-to-Global Memory. The global-to-global scenario of this experiment is depicted leftmost in Figure 1. A block or vector of compressed data is fetched from *global memory*. It then is decompressed by using bit-unpacking or GPU-FOR decompression, and the decompressed data (1GB) is directly written back to *global memory*. We repeat this experiment for every bit-width between 1 and 32. We bypass the L1 cache and make no use of *shared memory*, but store the results directly in global memory. The results in Table 1 show that on the Tesla T4, FastLanes outperforms Tile-based for both bit-unpacking (GPU-FOR vs FLS-BP) and DELTA decoding (GPU-DFOR vs FLS-DELTA). On the V100, both FastLanes and Tile-based are memory bandwidth bound, hence they have an *exactly* similar execution time.

Global-to-Shared Memory and `SUM`. We now fetch compressed data from *global memory*, decompress the data to *shared memory*, and directly perform a SUM on the decompressed data (middle of Figure 1). This aggregation is necessary to prevent the CUDA compiler from optimizing away all computation. The result of this local aggregation in shared memory is then written again to *global memory*, such that the aggregation value persists between the execution of multiple blocks. This avoids two passes to global memory: first decompressing and writing the data to *global memory* and then, when decompressing is finished and all data is stored, fetching this data again from *global memory* to perform any operations on the decompressed data.

Performing a "simple" aggregation now includes (1) an explicit allocation of shared memory for every block and (2) an aggregation in each thread in the corresponding thread block and a write to global memory for each block that is executed. Explicitly using shared memory incurs overhead, especially for Tile-Based, since it otherwise only uses GPU registers. However, if one stores intermediate results in registers (which is thread-local memory), this intermediate result is not accessible by other threads. The aggregation in Tile-Based ultimately requires 128 threads to each aggregate 4 values, which incurs 128 writes per 512 values to store the temporary result in shared memory. Finally, the final aggregation is written back to global memory.

For FastLanes, global-to-shared works by decoding a vector of 1024 values using one warp, such that each of the 32 threads decodes and directly aggregates 32 values. This results into 32 writes for the intermediate result into shared memory and one write to global memory for each 1024 values. By avoiding to write the large uncompressed (32-bits) results back to global memory, all kernels are now compute-bound. This reveals that FastLanes indeed is computationally faster than Tile-based compression. A somewhat unexpected result on T4 is that the Tile-Based kernels are in fact slower than in the global-to-global benchmarks for GPU-FOR and similar behavior is observed for GPU-DFOR on the V100 (1.77 vs 1.90 ms), which is depicted in Table 2. This increase in compute can be attributed to the fact that including a SUM is computationally intensive, and since GPU-DFOR seems compute bound instead

Table 4: *global-to-shared* memory + `SUM` with varying bit-width for Tile-based (GPU-FOR) and FastLanes (FLS-BP)

	GPU-FOR		FLS-BP	
Bits	T4	V100	T4	V100
1	10.09 ms	1.21 ms	2.34 ms	0.435 ms
3	10.02 ms	1.22 ms	2.42 ms	0.434 ms
4	10.09 ms	1.22 ms	2.45 ms	0.436 ms
8	10.14 ms	1.23 ms	2.60 ms	0.437 ms

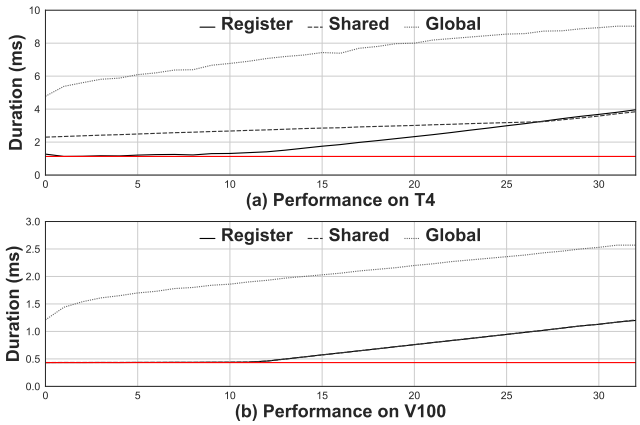

Figure 4: Micro-benchmarks for FastLanes unpacking into 32-bits values. X-axis is the bit-width of the compressed column (256M values). The red horizontal line is estimated computational cost of unpacking. The solid black line (global-to-shared) shows the impact of read-bandwidth; the dotted line the impact of write-bandwidth (global-to-global).

of memory bandwidth bound on both T4 and V100, this leads to deteriorating performance on both GPUs.

In Table 4 we confirm that FastLanes is consistently 3-5x faster than Tile-Based in global-to-shared decoding, for various bit-widths. Its kernels are compute-bound, as they only read compressed data, and the execution time increases by adding an aggregation. Increasing the data volume does not affect performance.

Global-to-Register. A third option is to *directly* store the output of the decompression in GPU registers, as opposed to *shared memory*. In CUDA, scalar variables are stored in registers by default by the compiler. Figure 4 shows the performance of FLS-BP for all bit-widths (1-32), using all three approaches. The red line models what we think is the computational cost of global-to-register. The solid black line (overall global-to-register time) follows this line for lower bit-widths, but starts to follow a linearly increasing line that we attribute to read cost from global memory. Note that the shared memory in V100 appears faster than on T4 as its line is completely identical on V100 with global-to-register. Global-to-global is at a constant distance from the read-cost, due to its additional cost for writing 1GB of data to global memory, which on V100 corresponds to the read-cost measured at 32-bits.

Measuring Compute. For the next micro-benchmarks, where we aim to measure the "raw" compute, we repeat the procedure described at the global-to-global memory benchmarks with one modification: we write nothing back to global memory. However, if we write nothing back to global memory the compiler optimizes everything away. Therefore, we trick the compiler by implementing an if statement containing a `STORE`, which has an almost 100% probability to evaluate to false. Now, we are able to measure the compute with increased precision. Note that the latency of fetching the compressed data from global memory is still included in the execution time.

Table 5: Measuring *compute* – thus no writes to global memory. Decoding of 3 to 32 bits (256M values) on Tesla T4. Compute is the "raw" compute of each method, including fetching compressed data from global memory. To-global are the values for T4 reported in Table 1, to highlight the difference with- and without writing back to global memory

	GPU-FOR	FLS-BP	GPU-DFOR	FLS-DELTA
compute	3.48 ms	0.98 ms	10.70 ms	2.49 ms
to-global	7.80 ms	5.89 ms	12.36 ms	6.05 ms

Table 6: Measuring *compute* – thus no writes to global memory. Decoding of 3 to 32 bits (256M values) on V100 GPU. Compute is the "raw" compute of each method, including fetching compressed data from global memory. To-global are the values for V100 reported in Table 1, to highlight the difference with- and without writing back to global memory

	GPU-FOR	FLS-BP	GPU-DFOR	FLS-DELTA
compute	0.86 ms	0.43 ms	1.58 ms	0.45 ms
to-global	1.60 ms	1.60 ms	1.77 ms	1.64 ms

The results in Table 5 indicate that FLS-BP is around 3-4x faster compared to GPU-FOR, and FLS-DELTA is around 4-5x faster then GPU-DFOR on a T4. On the V100, this difference is around 2x for FLS-BP and GPU-FOR, and 3x for FLS-DELTA and GPU-DFOR (Table 6). Also, on T4, FLS-BP is 2-3x faster then FLS-DELTA in terms of compute, and the same phenomenon occurs for GPU-FOR and GPU-DFOR. For V100, the difference for FLS-BP and FLS-DELTA is minimal, indicating that it is close to the minimum compute. GPU-FOR is around 2x as fast compared to GPU-DFOR on V100. Remarkably, in the global-to-global memory benchmarks the execution times of FLS-BP and FLS-DELTA are very similar. An explanation for the smaller difference between global-to-global and solely compute for FLS-DELTA and GPU-DFOR is that the increased compute hides the write latency to global memory.

Investigating Occupancy. To explore whether we can further optimize FLS-BP global-to-shared and FLS-DELTA global-to-shared, we investigate the GPU utilization. We found that both FLS-BP and FLS-DELTA suffer from low occupancies compared to Tile-Based (Table 7). In addition, for both FLS-BP and FLS-DELTA the occupancy on V100 is structurally lower.

The lower occupancy on V100 can attributed to the fact that for both T4 and V100 GPUs there are 64k 32-bit registers available per SM, even though V100 provides double the amount of maximum active threads per SM (Table 3). As a consequence, software employed on the V100 suffers from high register pressure. Compiling with `ptxas=-v` shows that FLS-BP uses at most 64 registers per

Table 7: *Occupancy* reported by ncu of both FastLanes and Tile-Based for bit-unpacking of 3 bits and DELTA decoding. T indicates the maximum theoretical occupancy of this configuration, A indicates the achieved occupancy during execution.

Method	T4 T	T4 A	V100 T	V100 A
GPU-FOR	100%	94.16%	100%	93.99%
FLS-BP	46.88%	44.67%	34.38%	16.09%
GPU-DFOR	100%	95.94%	100%	96.15%
FLS-DELTA	21.88%	20.75%	17.19%	16.04%

Table 8: Micro-benchmarks for different configurations when decompressing 3 bits using FLS-BP. Both timing and occupancy are reported. Occupancy T indicates theoretical occupancy, Occupancy A indicates achieved occupancy. Experiments are done on Tesla T4.

Configuration	Exec. time	Occupancy T	Occupancy A
<32, 32>	2.42 ms	46.88%	44.67%
<64, 32>	2.24 ms	40.81%	43.75 %
<128, 32>	2.33 ms	37.50%	34.63%
<256, 32>	3.49 ms	25.00%	24.63%

thread. This leads to at most $64K/64 = 1024$ active threads; which equals the maximum amount of 1024 active threads per SM on T4 (Table 3). On V100, this is below the maximum amount of 2048; which already leads to a lower occupancy. This phenomenon is also visible in Table 7. However, the occupancy is below 50% in all cases. This means that there are other factors that attribute to the low occupancy. These are (i) there are too little blocks or warps launched per SM and (ii) the required amount of shared memory per block is too high.

Block Size. We benchmark different configurations for FLS-BP *global-to-shared*, to investigate whether increasing block size improves the occupancy reported in Table 7. However, Table 8 shows that although the execution time increases slightly for a block size of 64, which is probably due to the reduction of scheduling overhead, both the theoretical and achieved occupancy gradually decrease. The decrease of occupancy can be attributed to the fact that we configure around 65kb (as pointed out by ncu) of shared memory at our initial launch configuration for FLS-BP global-to-shared. This implies that it is not possible to achieve a higher occupancy in our current implementation since we are limited by the required amount of shared memory per block. It is therefore more favorable to use the global-to-register approach, where shared memory usage will not become a bottleneck for bit-unpacking.

5 FASTLANES ON CRYSTAL

Our basic implementation of FastLanes on Crystal is referred to as FLS-GPU. In FLS-GPU we integrate FastLanes with Crystal using vectorized decompression (shown in the second scenario of Figure 1), where we decode the 1024 values using a block consisting of 32 threads, i.e. a single warp, such that each thread decodes 32 values. This approach is similar FastLanes on CPUs and is thus trivial as a basic implementation. Crystal however uses registers

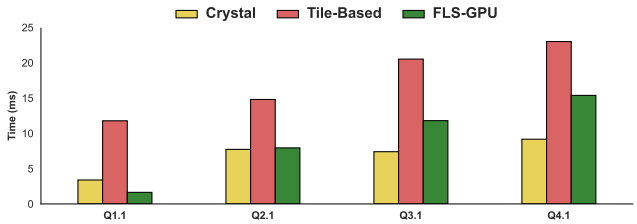

Figure 5: End-to-end SSB query execution times (SF10) on Tesla T4. Naive FLS-GPU significantly improves Q1.1 and generally performs better than Tile-Based decompression. Compared to Crystal however, there is still a performance penalty.

to store in-flight data. Relying on registers can be tricky, since the programmer does not have explicit control over the placement of data into registers. To guarantee that data resides in registers Crystal processes only 4 values per thread. Each thread in a warp can operate on at least 32 32-bit registers on a V100 GPU, which means that registers can be used for up to $\frac{32}{4} - 1 = 7$ columns in Crystal without a performance penalty.

When integrating FastLanes in Crystal, we choose to follow a similar approach which resembles the global-to-register approach explained in section 4. This is more beneficial since (i) profiling indicates that the occupancy is inevitably low due to shared memory usage being a limiting factor for the global-to-shared version of FLS-BP and (ii) the micro-benchmarks in Figure 4 show that the global-to-register approach is the most performant for data that is bit-packed in small bit-widths.

End-to-end benchmarks. We now benchmark the integration of FastLanes (de)compression in end-to-end queries, for which we use the Star Schema Benchmark (SSB) [6] queries implemented in Crystal [8]. SSB is a modified form of the TPC-H benchmark. Alongside integrating FastLanes in Crystal, we compare the benchmark results against Crystal-opt [2] and Tile-based decompression integrated in Crystal as reported in [9]. We only benchmark a single query from each different query family (Q1.*, Q2.*, Q3.* and Q4.* resp.), such that we can compare the performance among different types of queries within SSB. We chose to benchmark a scale-factor of 10 since this will fit as a whole in global memory.

Figure 5 shows that for all queries, FLS-GPU performs better compared to Tile-Based. However, except for Q1.1, FLS-GPU performs worse than baseline Crystal. This is because FLS-GPU incurs extra operations for decompression, and its naive approach of unpacking 32 values per thread with a maximum of 1024 values per block leads does not leverage the parallelism provided by GPUs.

Another unexpected result is the difference in behavior of FLS-GPU on the V100 and T4 as observed in figure 5 and figure 6. In fact, on T4, FLS-GPU performs significantly better than Tile-Based for all queries. On V100, however, Tile-Based outperforms FLS-GPU on Q3.1 and Q4.1. To find an explanation for this behavior we investigate whether register spilling or low occupancy cause this low performance. Specifically occupancy might be problematic, since Table 7 already indicated very low occupancy rates for FLS-BP and FLS-DELTA on a V100 GPU – which negatively affects performance.

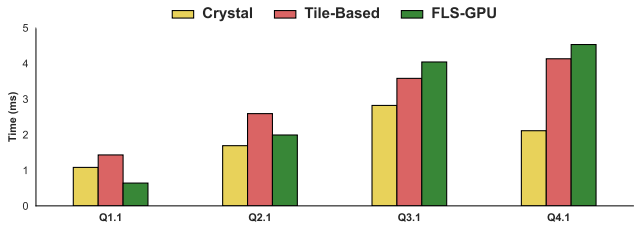

Figure 6: End-to-end SSB query execution times (SF10) on NVIDIA V100. Naive FLS-GPU still improves Q1.1 but generally performs worse compared to both Tile-Based decompression and Crystal.

Table 9: Occupancy for Q1.1, Q2.1, Q3.1 and Q4.1 using FLS-GPU. Both *theoretical* occupancy (`O-T`) and *achieved* occupancy (`O-A`) are reported for both T4 and V100 GPUs.

Query	O-T T4	O-A T4	O-T V100	O-A V100
Q1.1	50.00%	49.07%	37.50%	36.39%
Q2.1	50.00%	48.54%	25.00%	24.32%
Q3.1	25.00%	24.50%	12.50%	12.11%
Q4.1	25.00%	24.67%	12.50%	12.17%

Register Spilling. FLS-GPU unpacks 1024 values at a time, putting high pressure on registers. Therefore, a possible cause of slowdown for FLS-GPU is register spilling. If the spill is significant and the caches are full, this can cause a high slowdown (explained in section 2.1). To determine if spilling occurs at compile time, we compile the queries with the `'-ptxas-options=-v'` flag. We found that no register spilling occurs *at compile time* for both FLS-GPU and Tile-Based. However, for some queries, such as Q3.1 and Q4.1, which includes large hash tables and multiple columns, FLS-GPU assigns up to 226 registers per thread. This high amount of registers limits the number of threads (and thus warps) we can execute concurrently on a SM, slowing down performance. Due to the high amount of registers per thread, we obtain a low occupancy of 25% and even 12.50% for both queries on T4 and V100 respectively (Table 9).

Low Occupancy. Increasing the occupancy will, *in most cases*, lead to better performance. In its current form, FLS-GPU can only reach 50% of occupancy (Table 7 and Table 9), since it assigns only 32 threads, i.e. one warp, to a single block. Since the block limits on T4 and V100 are 16 and 32 with corresponding warp limits of 32 and 64 respectively, the highest theoretical occupancy we can achieve is 50% – assuming that register and shared memory usage are no limiting factors. On V100, where there are on average only 32 registers per thread available (Table 3), we only reach a very low theoretical occupancy of 12.50% for Q3.1 and Q4.1. This low occupancy explains the bad performance of FLS-GPU compared to Tile-Based on V100. To increase both theoretical and achieved occupancy, we consider increasing the thread block size from 32 to 128 or 256 such that we will achieve the maximum amount of blocks or warps that run concurrently on a SM.

However, only increasing the block size is not enough, as shown in Table 8. To achieve a better occupancy we therefore need to (i) reduce the amount of shared memory we use per block and/or (ii) decrease the amount of registers used per thread. Since FLS-GPU uses the global-to-register approach, we already minimized the amount of shared memory used for bit-unpacking. If we want to boil down shared memory usage even further, this would require adapting the SSB queries in `Crystal`, which is out of scope. The second option is to reduce the register usage by unpacking less values per thread. For example, we could unpack 8 values per thread using 128 threads per block, which still leads to a single block processing 1024 values. This form of scheduling almost resembles the Tile-based processing model, which unpacks 4 values per thread for GPU-FOR and GPU-DFOR, using a total of 128 threads per tile (i.e. thread block). Another way to reduce the registers per thread is compiling with the `-maxregcount` flag or using `__launch_bounds()` parameter to limit the amount of registers per thread for all or specific kernels. Forcing a lower amount of registers per thread however leads to register spilling, which becomes quickly very expensive and reduces the overall performance. Therefore, we aim to process less values per thread to reduce register pressure in a more natural way.

5.1 FLS-GPU-opt

The results in figure 5 and figure 6 show that FLS-GPU (green) only improves performance over Crystal (yellow) in Q1.1. The reason why this happens is that the other queries involve more columns and joins which require in-memory hash-tables. As a consequence, the register pressure generated by FLS-GPU becomes too high in these queries. The register pressure in combination with scheduling too few blocks per SM leads to a low occupancy which in this case severely affects performance. Therefore, we started considering methods to reduce the register pressure and increasing the thread block size, moving to FLS-GPU-opt; depicted in the rightmost scenario of Figure 1. The optimized version of FastLanes on Crystal, FLS-GPU-opt, addresses shortcomings of FLS-GPU while leveraging GPU parallelism. This includes releasing pressure on registers by processing *mini-vectors* and using *compressed execution*. In addition, FLS-GPU-opt achieves a better compression ratio by using RLE for suitable SSB columns, such as `lo_orderdate`. Each of the optimizations is briefly explained below.

Processing *mini-vectors*. To release pressure of registers and shared memory we partition a vector of 1024 values into *mini-vectors* of 256 values. This means that each thread in a warp now processes 8 values at-a-time, thus using 8 32-bit registers per column, a 4x reduction of register pressure. Technically, this means that bit-unpacking logic is split over 4 FastLanes unpack methods; each delivering 256 values. For bit-widths that are not multiples of 4-bits this leads to some additional work if the unpacking does not start aligned on a 32-bits values, but the extra effort is low.

Compressed Execution. While processing queries in Crystal, all SSB columns are handled in-flight as 32-bit integer values. However, some columns of the SSB benchmark can be encoded in significant smaller data types. Using smaller data types is beneficial to reduce both the memory footprint and memory bandwidth. This however is not natural to GPUs, since each register in a GPU spans one *word*, and each word consists of 32 bits. It is thus convenient to

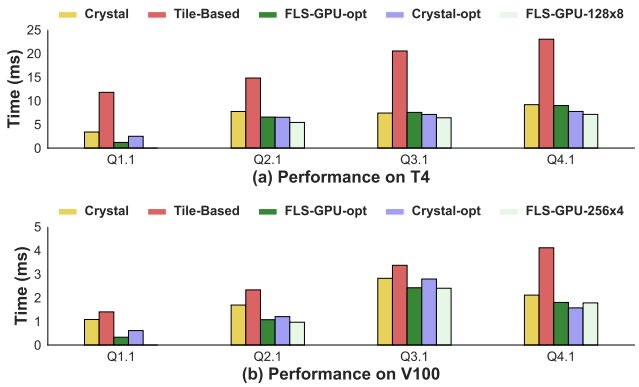

Figure 7: End-to-end SSB query execution times (SF10). Naive FLS-GPU significantly improves Q1.1; but it generates too much register pressure in the other queries, which involve more columns and are hash-probe rather than scan-bound. By reducing the decompression granularity with mini-vectors, using more threads per block and simulating RLE, FLS-GPU-opt can match its performance nevertheless.

decompress values into 32-bit integers to align with the word size. However, decompressing values into 32 bits is inefficient if significantly less bits are needed. For example, let's assume that we are able to represent values of a column in 8-bits. We then can *partially* decompress the bit-packed values into four 8-bit lanes in one 32-bit register, instead of decompressing a single 8-bit value into 32 bits. This also allows us to directly operate on these values at the same time, *within* a single thread. Thus, we use some SIMD parallelism with the GPU SIMT execution model. As a result, we are able to (i) fit more data into registers which avoids spilling to L1 cache (ii) enhance more data-parallelism by performing multiple operations at the same time and (iii) reduce bandwidth by a factor 4.

Predicate Pushdown. In real-world systems, columns that do not benefit from bit-packing will not be compressed. Therefore, we leave the column `extended_price` uncompressed, and we use the `<PredLoad>` predicate-pushdown optimization proposed by Crystal-opt [2] to reduce bandwidth. Crystal-opt [2] showed that Crystal loads unnecessary data from *global memory* and this affects performance.

Note that when using scans on compressed columns, such predicate-pushdown is impossible (or, it would require random access to compressed data). Therefore, *all* compressed data needs to be decompressed at least to *shared memory*, incurring global memory bandwidth. Only if an entire vector or tile would have zero selected tuples, this step could be skipped. This is similar to another optimization of Crystal-opt, which terminates a thread and eventually a warp early if no tuple is selected. However, this is mostly beneficial for highly selective queries, i.e. when a chunk of values does not satisfy any of the selection flags. None of the SSb queries Q1.1, Q2.1, Q3.1 and Q4.1 benefit from this.

Simulating RLE. In the current port of FastLanes to GPU, we do not support RLE yet. The SSB LINEORDER table is clustered on order, which means that it is quite RLE-compressible, as all columns that contain order information, rather than lineitem information, repeat on average four times. In the SSB queries tested here, this concerns the `lo_orderdate` and `lo_custkey columns`. Lacking RLE, the compression ratio FLS-GPU achieve is diminished to about 1.5x. In real-life datasets, such as public BI [4], FastLanes can achieve a compression ratio of 8X. To mitigate the bad compression ratio, partially caused by our lack of an RLE implementation, we decided in a separate experiment to sort LINEORDER on columns `lo_orderdate` and `lo_custkey`. This allows to store `_orderdate` in 8 bits instead of 16, and `l_custkey` in 8 bits instead of 20 (real RLE would reduce this even to 6 + 8 bits).

Table 10: Occupancy for Q2.1, Q3.1 and Q4.1 using FLS-GPU-128x8 for T4 and FLS-GPU-256x4 for V100. Both *theoretical* occupancy (`O-T`) and *achieved* occupancy (`O-A`) are reported. Q1.1 is not included, since this query already outperformed Crystal, Tile-Based and Crystal-opt significantly.

Query	O-T T4	O-A T4	O-T V100	O-A V100
Q2.1	100.00%	93.23%	100%	92.25%
Q3.1	87.50%	81.49%	75.00%	68.90%
Q4.1	87.50%	80.92%	75.00%	67.75%

Larger Block Size. Table 9 indicates that the occupancy of FLS-GPU on both T4 and V100 is low for all queries, but particularly for Q3.1 and Q4.1. To improve occupancy by lowering register pressure, we now move to a 8-values-per-thread model, which we call *mini-vectors*, as described above. Instead of only launching thread blocks consisting of 32 threads, we now increase the size to 128 threads per block for T4 to still decode 1024 values per block (FLS-GPU-128x8). This allows to execute more warps concurrently while reducing register pressure. For V100 however, there are even less registers available per thread, leading to a higher register pressure. Therefore, we choose to use a 4-values-per-thread model, using 256 threads per block to decode 1024 values per block (FLS-GPU-256x4). For these configurations, we also remove the predicate pushdown optimization and instead compress all columns.

5.2 Discussion

SSB Q1.1 is a simple scan with filter and aggregation. The roof-line analysis in [2] already showed that this query is the most scan-bound – and thus stands to profit most from compressed storage. The fact that Tile-Based compression is not able to improve performance of this query is a missed opportunity, but explainable from the fact that its encoding format lacks data-parallelism needed by GPUs. The interleaving of values in a vector employed by FastLanes however allows Q1.1 to execute 2-3x faster, illustrated also in Table 11. For Q1.1 the predicate-pushdown on `extended_price` provided FLS-GPU-opt most of the additional gains over FLS-GPU. For the other queries, where FLS-GPU suffers from too high register pressure, the FLS-GPU-opt benefits most from using mini-vectors. Notably, we did not manage (yet) to make compression faster using the idea of compressed execution, i.e. using data types smaller than 32-bits. The reason for this lack of success is as of yet unclear, and there are still techniques we could try. We further think that more

Table 11: On the scan-bound Q1.1 that stands to profit most from compressed scans, FLS-GPU shows strong performance, which is significantly enhanced in FLS-GPU-opt.

Scheme	SF1-T4	SF10-T4	SF1-V100	SF10-V100
Crystal	0.35	3.39	0.115	1.080
Crystal-opt	0.26	2.49	0.070	0.608
FLS-GPU	0.21	1.92	0.087	0.642
FLS-GPU-opt	0.139	1.19	0.057	0.335

complex encoding schemes, like RLE and DICT, which we so far have not implemented, could benefit from GPUs.

For Q3.1 we managed to increase the performance further by sorting LINEORDER on `lo_orderdate` and `lo_custkey` to achieve a better compression ratio. The query performance from FLS-GPU-opt goes from 8.17 ms to 7.54 ms on T4, and from 2.78 to 1.33 on V100. Specifically for the V100 GPU the performance increase is a factor of 2, which is significant. For Q4.1, the improved compression ratio provided by sorting did not have a significant impact. We do intend to re-benchmark FLS-GPU when RLE support is ready and this sorting is no longer required.

Lastly, aiming to increase occupancy, we scheduled larger thread blocks. We found that for the T4 high occupancy is achieved for thread blocks of 128 threads, that process 8-values-per-thread (Table 10). For Q2.1 this improved the execution time from 6.55 to 5.42ms, for Q3.1 from 7.54 to 6.40ms and for Q4.1 from 8.99 to 7.12 ms. For V100, we still suffer from severe register pressure, and therefore were not able increase the occupancy with the 128x8 format. Instead, we tried 256x4 to process even less values per thread. However, register pressure remained problematic for the occupancy – only little performance improvement is observed. We note though, that Q3.1 and Q4.1 which involve more columns than the other two queries, also cause lower occupancy for Crystal itself.

6 CONCLUSIONS AND FUTURE WORK

In this paper, we tested the data-parallel layout of FastLanes on GPUs. Both our micro-benchmarks as well as end-to-end SSB query results show encouraging results. The micro-benchmarks in section 4 showed that FLS-GPU outperforms Tile-Based decoding by a factor of 3-4x for bit-unpacking against GPU-FOR and FLS-DELTA decoding against GPU-DFOR. We also show in contrast to Tile-Based (which causes a 35% slowdown of end-to-end queries), that the overhead incurred by FastLanes decompression in Crystal is offset by reduced memory bandwidth; an important bottleneck for data processing on GPUs.

We also found drawbacks of the original 1024 tuples at-a-time decoding granularity of FastLanes: this forced it to use at least 32 registers per thread - which are not always available, or to store 1024 values on shared memory for each block. This proved to become a bottleneck on more complex queries with a multiple columns to process. We addressed this issue using the idea of *mini-vectors* and larger thread blocks, which perform FastLanes decompression in four steps of each 256 or 128 values to reduce pressure on GPU registers and shared memory, as well as the idea of decoding into thin data-types (8- and 16-bits). We however experienced that register pressure on the V100 remains a challenge, and were not able to significantly improve its execution time using a 256x4 configuration.

FastLanes on GPUs is still in an early stage of development, and its more complex encoding schemes (e.g. RLE) were not available yet in CUDA during these experiments. This causes SSB experiments to experience lower compression ratio's than are normally possible with FastLanes. The experiments with an artificially enhanced compression ratio (by sorting the LINEORDER table) already show that end-to-end query performance will further improve once the FastLanes GPU implementation reaches greater maturity.

6.1 Future work

Improving Mini-Vectors. In FastLanes, we bit-pack 1024 tuples using the interleaved layout, which has as advantage that all 32 threads do the same decoding work (no divergence) and have coalesced memory access. To support access using mini-vectors (we experimented with 8 resp. 4 values per thread), for bit-widths other than multiples of 4 resp. 8, memory access is not 32-bits aligned. In our experiments we used our original decoding methods and mitigated by rounding up bit-widths to the closest higher multiple of 4 resp. 8, hurting compression ratio and thus performance. This rounding up can be avoided by a proper implementation for all bit-widths at some additional computational cost during decoding.

Reducing Mini-Vectors. The observed strong effects of register pressure make us consider even smaller mini-vectors, and even the extreme approach of threads doing single-tuple access. The trade-off here is increased computational overhead in decoding calls, for invoking the decoding action appropriate for each mini-vector, as well as for interpreting cascaded encodings. The decoding interpretation cost would in the extreme case be incurred for each tuple. We note that the variability of data in-the-wild requires an interpreted approach for decoding, as parameters and encodings used will vary between different parts of a column.

Adding DELTA and RLE In this paper, we mainly focused on bit-packing, as FastLanes on GPU is still in a very conceptual phase of development, and the full set of basic encodings had not yet been ported to CUDA (specifically DELTA/RLE and DICT). This is an opportunity to further improve our results, since quite a few columns from the SSB benchmark can be better compressed by RLE. We think that using RLE we can further speed up SSB queries as the overall compression ratio would increase.

Compressed Execution. We observed that the performance bottleneck of SSB queries Q2.*, Q3.*, and Q4.* are join probes. Therefore, memory latency is a significant cost, caused by the random and non-coalesced nature of hash-lookups; which may only be alleviated by caching (mainly in the L2 cache). While this problem appears to be unrelated to our main topic of accelerating compressed scans from a novel big data format, we do think that the idea of compressed execution (decompressing to thinner types) could allow to build smaller hash tables (which are faster hash tables thanks to improved caching locality). Further, a GPU data processing engine could potentially even squeeze in-flight data, by storing multiple thin (e.g., 8- or 16-bit) values in a single 32-bit value, to reduce register or shared memory pressure; thereby enabling higher kernel performance.

REFERENCES

[1] Azim Afroozeh and Peter Boncz. 2023. The FastLanes Compression Layout: Decoding> 100 Billion Integers per Second with Scalar Code. *Proceedings of the VLDB Endowment* 16, 9 (2023), 2132–2144.

[2] Jiashen Cao, Rathijit Sen, Matteo Interlandi, Joy Arulraj, and Hyesoon Kim. 2023. GPU Database Systems Characterization and Optimization. *Proceedings of the VLDB Endowment* 17, 3 (2023), 441–454.

[3] Wenbin Fang, Bingsheng He, and Qiong Luo. 2010. Database compression on graphics processors. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 670–680.

[4] Maximilian Kuschewski, David Sauerwein, Adnan Alhomssi, and Viktor Leis. 2023. BtrBlocks: Efficient Columnar Compression for Data Lakes. *Proc. ACM SIGMOD* 1, 2 (2023), 118:1–118:26.

[5] David Luebke. 2008. CUDA: Scalable parallel programming for high-performance scientific computing. In *2008 5th IEEE international symposium on biomedical imaging: from nano to macro*. IEEE, 836–838.

[6] Patrick E O'Neil, Elizabeth J O'Neil, and Xuedong Chen. 2007. The star schema benchmark (SSB). *Pat* 200, 0 (2007), 50.

[7] Eyal Rozenberg and Peter Boncz. 2017. Faster across the PCIe bus: a GPU library for lightweight decompression: including support for patched compression schemes. In *Proceedings of the 13th International Workshop on Data Management on New Hardware*. 1–5.

[8] Anil Shanbhag, Samuel Madden, and Xiangyao Yu. 2020. A study of the fundamental performance characteristics of GPUs and CPUs for database analytics. In *Proceedings of the 2020 ACM SIGMOD international conference on Management of data*. 1617–1632.

[9] Anil Shanbhag, Bobbi W Yogatama, Xiangyao Yu, and Samuel Madden. 2022. Tile-based lightweight integer compression in GPU. In *Proceedings of the 2022 International Conference on Management of Data*. 1390–1403.

[10] Johannes Unterguggenberger, Bernhard Kerbl, and Michael Wimmer. 2023. Vulkan all the way: Transitioning to a modern low-level graphics API in academia. *Comput. Graph.* 111 (2023), 155–165.

[11] Xinyu Zeng, Yulong Hui, Jiahong Shen, Andrew Pavlo, Wes McKinney, and Huanchen Zhang. 2023. An Empirical Evaluation of Columnar Storage Formats. *arXiv preprint arXiv:2304.05028* (2023).