Improving GPU Matrix Multiplication by Leveraging Bit Level Granularity and Compression

Johannes Fett[1] Christian Schwarz[1] Urs Kober[1] Dirk Habich[1] Wolfgang Lehner[1]

Abstract: In this paper, we introduce BEAM as a novel approach to perform GPU based matrix multiplication on compressed elements. BEAM allows flexible handling of bit sizes for both input and output elements. First evaluations show promising speedups compared to an uncompressed state-of-the-art matrix multiplication algorithm provided by Nvidia.

Keywords: GPU; Matrix multiplication

1 Introduction

GPUs are becoming increasingly more popular for data analytics and compute workloads with increasing memory demands. GPUs share the constraint of having a significantly smaller memory capacity compared to CPUs with DRAM. One approach to mitigate this issue is to use compression. Our focus is to explore how to perform calculations on already compressed data.

In this work, we introduce BEAM (bitwise efficient matrix multiplication), a novel concept to directly compute on compressed elements in GPU memory. Instead of using native data types, we offer bit level granularity for unsigned integer based matrix multiplications. BEAM calculates directly on compressed data on a bit level granularity. Different problems that arise from bit level computation on GPU are discussed and strategies to deal with them introduced and evaluated. This paper focuses on unsigned integer based matrix multiplication to demonstrate that even in unfavourable compute bound use cases, compression can still be beneficial. In Section 2 preliminaries about compression on GPU and GPU architecture are introduced. Section 3 focuses on a detailed description of BEAM and strategies dealing with overflows and calculation of output bit sizes. Section 4 deals with related work and section 5 summarises the contribution.

[1] TU Dresden, Database Research Group, Nöthnitzer Str. 46, 01187 Dresden, Germany {johannes.fett|christian.schwarz5|urs.kober|dirk.habich|wolfgang.lehner}@tu-dresden.de

doi:10.18420/BTW2023-49

2 Preliminaries

2.1 GPU Architecture

A Nvidia GPU consists of a large number of arithmetical logical units called CUDA cores. Groups of 64 CUDA cores form a functional block called streaming multiprocessor. A streaming multi processor also shares register memory and shared memory across all its cores. A shared L2 Cache and VRAM (global memory) is accessible by all streaming multiprocessors through a shared memory bus system. The total amount of global memory is up to 80 GB for current GPU generations. Shared memory per streaming multi processor ranges from 48 KiB to 64 KiB depending on the GPU generation. The programming Model of a GPU is called single instruction multiple threads. A large number of threads is spawned to perform a computation (kernel). A group of threads is called a block. The total amount of threads is partitioned into a number of blocks. Each block is assigned to streaming multiprocessor. Most instructions are performed in groups of 32 threads at once, which is called a warp.

2.2 Compression on GPU

Typically, integer calculations work on an element level granularity. With BEAM we introduce the ability to calculate elements on a flexible bit level granularity. Specifically, we demonstrate a matrix multiplication on compressed elements. Different approaches to integer data compression have been covered by [Da17]. By combining different compression algorithms, an improved compression rate can be achieved. However, in our experiments we assume that all elements are compressed by zero suppression.

3 BEAM

BEAM allows flexible matrix multiplications by allowing elements on a bit level granularity instead of typical byte based data types. In case of using zero suppression, empty bits can be removed from Integer based data types. Test data is generated accordingly to conduct experiments on different bit sizes of elements ranging from 1 to 64 bit per element. The input bit size is static across the elements within a matrix to avoid the need for a prefix sum to access data elements. The supported data format is only unsigned Integers between 1 and 64 bit size.

3.1 Output Bit Strategies

The desired output bit size can vary depending on different strategies. If statistical information of a matrix is known, there might be a lower possible output bit size that fits all elements. If

Strategy	Description	Acronym
Same as input	output bits same as input bits	sai
Ceil to Power of Two	output bits is the smallest power of two greater or equal to the input bits	c2p2
Max Value	largest required output value, maximum 64 bit	maxv

Tab. 1: An overview of different strategies for calculating the output bit size.

Strategy	Description	Acronym
Overflow	default CUDA behavior: wrap around zero and cycle through the value range	ovf
Saturate	Stay at the largest possible value. This behavior mimics the common Sigmoid activation function from machine learning applications	sat

Tab. 2: An overview of different strategies for dealing with overflows

Memory layout	Description
Canonical Layout (naive)	One matrix element uses 64 bits in memory.
Tight types	A bit level element is packed into a single next largest native datatype
Padded Slabs (slabs)	As many complete matrix elements as possible are placed into one 64 bit slab
Tight Packing (nogaps)	Memory is viewed as a contiguous bitstream

Tab. 3: Memory representations for matrix multiplication.

there is no available information, defining a maximum needed bit size to avoid an overflow is a safer approach.

3.2 Overflow Behavior

In the case the calculated element in the output matrix causes an overflow, our approach offers two different strategies to deal with overflows. Saturate will pin the result at the max value of the value range in case of an overflow. This is realized by a builtin GPU intrinsic. Overflow is the default CUDA behavior that will lead to values cycling through the value range in case of an overflow.

3.3 Matrix Memory Representation

Because the input (and output) bit length of elements is not fixed to powers of two, using the native C integer types is inefficient. Therefore we experiment with different memory layouts to increase the performance. The matrices are stored row-major, without loss of generality.

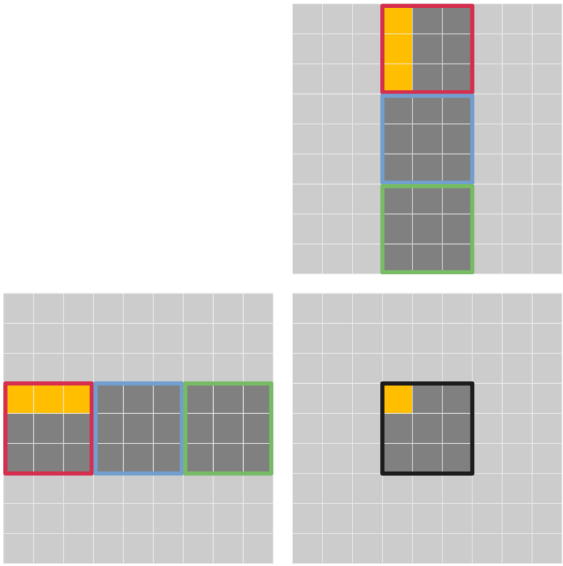

Fig. 1: Squarestride Matrix Multiplication example

For an overview of memory representations see Table 3. Slabs tries to fit as many bit level elements as possible into one 64 bit element. While this simplifies handling the memory it also leads to internal fragmentation. For example, 8 different 8 bit elements can fit without fragmentation into one 64 bit element. However, in case of 25 bits per element, only two will fit and consume 50 bits of space. The remaining 14 bits remain empty and lead to fragmentation. To avoid fragmentation the nogaps approach has been developed. In this case a contiguous bitstream is used to store all compressed elements. To allow simpler computations, end-of-row padding is introduced to the next 64 bit multiple.

3.4 Algorithms

This section contains an overview of all evaluated matrix multiplication approaches.

Squarestride Every thread block is responsible for exactly one sub-block of the output matrix and for every one of its threads for exactly one element within it (black outlined 3x3 box in the image 1). Both, this sub-block and the currently required sub-blocks for the left and right matrix, are kept in shared memory to reduce redundant reads from global memory. This sub-block based computation works such that sub-blocks are loaded one by one going inwards. Start by loading both red outlined sub-blocks and do partial computation, then continue to load both green sub-blocks. This pattern repeats until all results values have

been calculated. If the matrix is not a multiple of the sub-block dimensions, the excess values are assumed 0, such that they stay neutral to the computation. This avoids additional bounds checks.

Flex out Because access and computation logic calculation is in some cases significantly more complex if support for variable number of output bits (unequal to the input bits) is added, both version are provided for fair comparison. The more flexible version is suffixed with flex out.

Baseline Guide This kernel uses the squarestride strategy and the naive memory layout. This is the approach described in the Nvidia programming guide matrix multiplication example 2 [Nv].

Tight Types Tight types is using the squarestride strategy. Input bitsize is rounded up to the next native datatype. For example a 31 bit element would be packed into one 32 bit integer.

Squarestride and Slabs This kernel uses the squarestride strategy and the slabs memory layout. The slabs memory layout is used in the shared memory sub-blocks as well. In this case one thread no longer handles one element of the output matrix but one slab. Because that would make the sub-blocks rectangular by the increased number of vertical slabs required, each thread handles as many rows as there are elements in one slab.

Squarestride Nogaps and Shared Memory Slabs This kernel uses the squarestride strategy and the nogaps memory layout. It is very similiar to the squarestride kernel with the difference of using the noslabs memory layout in global memory. For faster computations, the slabs memory layout is still used in shared memory. This also avoids stitching together an element from two slabs during the computation within the individual sub-blocks.

Squarestride Flex Out This kernel is a version of matrix mul squarestride that supports a variable number of output bits. To achieve this, the number of elements stored in each shared memory slab is reduced to the number of elements in an output slabs. Note that the bit size within the shared memory is still only following the input bit count. Flex out means, that different output strategies are covered by the same kernel.

GPU	GPU Generation	CMake	CUDA	NVCC	G++
RTX 8000 Quadro	Turing	3.25.1	11.5	11.5.119	9.3.0-17

Tab. 4: An overview of used build and compilation tools as used in the evaluation.

Squarestride Nogaps Shared Memory Slabs Flex Out This kernel is a version of matrix mul squarestride nogaps that supports a variable number of output bits. This is achieved using the same adaption as described by flex out. Slabs are being held in shared memory.

4 Evaluation

In this section, all of BEAMS algorithms are compared against the state-of-the-art Nvidia matrix multiplication. While all approaches receive the same input data, Nvidias approach works on an element level granularity with naive memory layout mentioned in 3. To allow a fair comparison, a variant of Nvidias matrix multiplication algorithm has been created that supports the saturated overflow behavior 2. As a rising trend, both general data sizes and machine learning models are growing in size. Thus, we evaluate both a small 64 MiB matrix multiplication and a larger 1 GiB matrix. Each data element is generated with varying bit size ranging from 1 to 64 bit. Bit sizes smaller than 64 bit use zero suppression to create a compressed data element. The Nvidia approach uses each element as 64 bit element. All experiments run with 3 repetitions. The average run time of all three is used in the evaluation. All experiments run on CUDA Cores and no tensor cores are used. Cublas is not used as frame of reference, as it does not support 64 bit Integer Operations. As BEAM is designed for Integer calculations, comparing against Cublas would be unfair.

4.1 Implementation

All experiments have been implemented in CUDA and C++. As build system Cmake is used. The Code builds with Clang and Nvcc. The experiments have been performed on an nvcc based binary.

4.2 Experimental setup

All experiments have been conducted on a Nvidia RTX 8000 GPU. For an overview of the system see Table 4. The GPU offers 48 GB GDDR6 Memory with a total theoretical maximum bandwith of 672 GB/s.

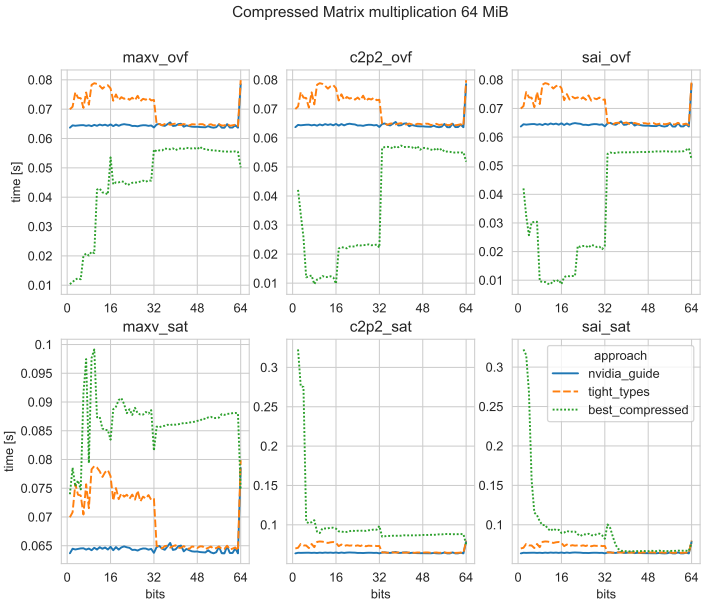

Fig. 2: 64 MiB Matrix multiplication. Compressed bitwise algorithms compared against Nvidia Matrix Guide and tight types approach. The best performing compressed algorithm is picked per point.

4.3 Results

Figure 2 shows an overview of different experiments on a 64 MiB data set. The grid of images is based on the chosen output bits approach in X direction as described in Table 1. The Y axis of the grid shows both different overflow approaches as shown in Table 2. Three different approaches are shown in each graph. Tight types and Nvidia guide have been explained in Section 3.4.. Best compressed is the best performing compressed algorithm per bit from a pool of all mentioned algorithms in Section 3.4.

In case of saturated overflow behavior, Nvidia's Matrix algorithm shows the best performance. Tight types performs better than best compressed across all three different bit output strategies. For saturated overflow behavior on 64 MiB Matrices, Nvidia's approach is the best. However, this changes if the overflow behavior is allowing overflows instead of saturating. In cases where the matrix multiplication will not overflow because the output data size is sufficiently large, the best compressed approach outperforms both, the Nvidia approach and tight types massively.

The size of data sets is constantly growing for machine learning and data management. To accommodate this trend, another experiment has been conducted on matrices with the size of 1 GiB. The experiment follows the same approach as the 64 MiB one. The results are

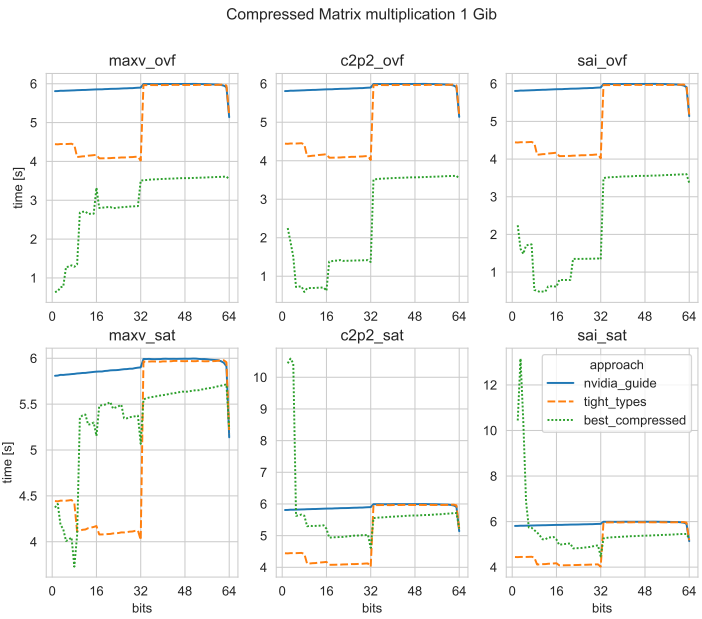

Fig. 3: 1 GiB Matrix multiplication. Compressed bitwise algorithms compared against Nvidia Matrix Guide and tight types approach

shown in Figure 3. In case of overflow behavior allows overflow, an increased speedup compared to the 64 MiB experiment can be achieved.

Saturated 1 GiB For saturated overflow behavior the result is vastly different and the Nvidia approach is the overall worst performing one. For the bit range of 33 to 64, tight types mirrors the Nvidia guide behavior, while best compressed outperforms both of them. The reason for this behavior is that tight types is only able to put one compressed element into one natively supported data type. Thus, a 33 bit unsigned integer leads to a 64 bit unsigned integer in case of tight types. Below 33 bits, tight types offers a speedup of about 1.4x compared to Nvidia's approach. For both output bit behaviors ceil two power two and same as input combined with saturated overflow behavior, tight types is the best solution for elements ranging from 1 to 32 bit size. In case of maximum value output behavior, the compressed approach offers a small speedup between 1 und 7 bits input bit size. In case of 64 bit, no compression occurs, which leads to similar results for all three approaches. For very low bit sizes, *best compressed* performs increasingly worse due to being more compute intense on a rising number of elements.

Overflow allowed 1 GiB Changing the overflow behavior to allow overflows, reduces the compute intensity of the algorithm. Instead of using an intrinsic to check for overflow and handling it with branching within the kernel, this approach is default GPU behavior and

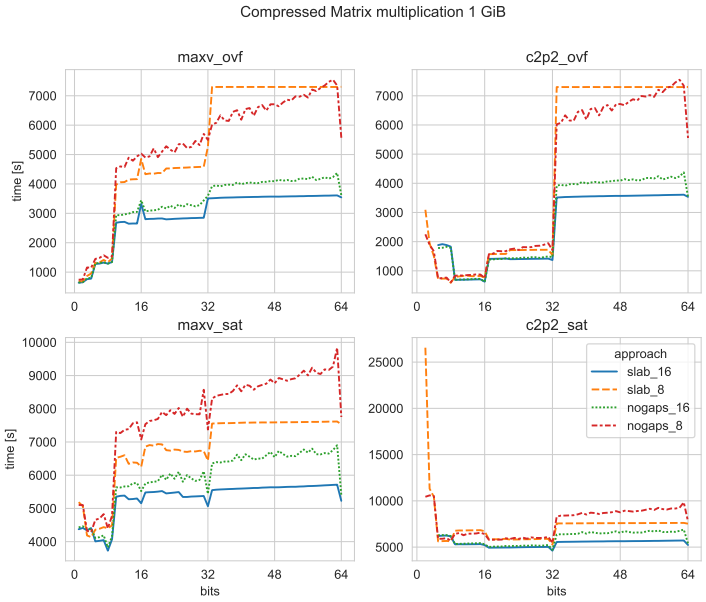

Fig. 4: 1 GiB Matrix multiplication. Comparison of different compressed computation algorithms

does not need any extra branches. Best compressed performs best across all input bit sizes with no exception. For larger output sizes in case of maximum value as chosen output bit strategy, best compressed performs slower than the other output bit strategies.

Comparison of different compression approaches Figure 4 shows 4 different compression algorithms, that are compared in detail. As the output bit strategy *same as input* is very similar to *ceil to power two*, it has been omitted from the graphs. Slabs is fitting into one element while ogaps features a contiguous bitstream. The missing data points indicate, that some CUDA configurations demand more shared memory than available. Overall the slabs approach outperforms nogaps with few exceptions. Within a block threads are 2 dimensional. Slab 16 for example uses 16 threads in two dimensions, which results in 256 total threads. Slab 8 only uses 8 threads per dimension, resulting in 64 total threads per block.

5 Related work

Shabag et. al have designed a compression framework integer based gpu computing [Sh22] called *tile-based lightweight integer compression*. It is based on storing compressed data in global memory and decompresses the data in shared memory. After computations, the data needs to be re-compressed and written back to global memory. The key difference to BEAM is, that BEAM does not need to decompress data before using it. However our

approach is limited to zero suppression, while Shabag et. al combine a number of different compression schemes to achieve higher compression rates. Also, their approach is focused on database operators and does not support matrix operations. Fang et. al have proposed compression for CPU GPU co-processing for databases [FHL10]. In this scenario, PCI-E becomes a major bottleneck, which can be improved by transferring compressed data.

6 Conclusion and summary

Our approach allows GPU matrix multiplication on a bit level granularity instead of the usual data element level granularity. It has been evaluated on different overflow strategies and output bit strategies. BEAM outperforms Nvidia slightly in case of a 64 MiB matrix and massively on a larger 1 GiB matrix. On average our approach offers a good speedup compared to the state-of-the-art approach. We look forward to further extend our bitwise GPU computation approach to other domains.

References

[Da17] Damme, P.; Habich, D.; Hildebrandt, J.; Lehner, W.: Insights into the comparative evaluation of lightweight data compression algorithms. Algorithms 1/1oranN, 1mappingdependingontheimplementation, 2017.

[FHL10] Fang, W.; He, B.; Luo, Q.: Database compression on graphics processors. Proceedings of the VLDB Endowment 3/1-2, pp. 670–680, 2010.

[Nv] Nvidia Matrix Multiplication Guide, `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#shared-memory`.

[Sh22] Shanbhag, A.; Yogatama, B. W.; Yu, X.; Madden, S.: Tile-based Lightweight Integer Compression in GPU. In: Proceedings of the 2022 International Conference on Management of Data. Pp. 1390–1403, 2022.