The Tensor-Core Beamformer: A High-Speed Signal-Processing Library for Multidisciplinary Use

Leon Oostrum[1], Bram Veenboer[2], Ronald Rook[3], Michael Brown[4], Pieter Kruizinga[4], John W. Romein[2]
[1]Netherlands eScience Center, Amsterdam, the Netherlands, l.oostrum@esciencecenter.nl
[2]ASTRON (Netherlands Institute for Radio Astronomy), Dwingeloo, the Netherlands, {veenboer, romein}@astron.nl
[3]Sioux Technologies, Eindhoven, the Netherlands
[4]Erasmus Medical Center, Rotterdam, the Netherlands

***Abstract*—Beamforming is a well-known technique to combine signals from multiple sensors. It has a wide range of application domains. This paper introduces the Tensor-Core Beamformer: a generic, optimized beamformer library that harnesses the computational power of GPU tensor cores to accelerate beamforming computations. The library hides the complexity of tensor cores from the user, and supports 16-bit and 1-bit precision. An extensive performance evaluation on NVIDIA and AMD GPUs shows that the library outperforms traditional beamforming on regular GPU cores by a wide margin, at much higher energy efficiency. In the 16-bit mode, it achieves over 600 TeraOps/s on an AMD MI300X GPU, while approaching 1 TeraOp/J. In the 1-bit mode, it breaks the 3 PetaOps/s barrier and achieves over 10 TeraOps/J on an NVIDIA A100 GPU. The beamforming library can be easily integrated into existing pipelines. We demonstrate its use for medical ultrasound and radio-astronomical instruments.**

***Index Terms*—Graphics Processing Unit, beamforming, ultrasound, radio astronomy**

I. Introduction

Beamforming is one of the basic ways to combine the signals from multiple sensors. It is a well-known technique to increase sensitivity in specified directions (the beams), and suppress signals from other directions. Beamforming can be used for sound and radio waves, for transmissions and reception, and can be performed on analog and digitized signals. The applications range from wireless communication to seismology.

This paper focuses mostly on digital beamforming. Depending on the application, the data rate of signals, the number of sensors, or the number of beams can be large. This leads to a compute challenge, while the signals typically need to be processed in real time, and often under constraints that limit the energy use.

In this paper, we explore the use of *tensor cores* for beamforming. Tensor cores are hardware matrix-matrix multiplication units found in most modern Graphics Processing Units (GPUs). They perform these multiplications much more efficiently than regular GPU cores, but typically only work for limited-precision input data (e.g. 8-bit integers), while the output is usually 32-bit integer or floating point. Tensor cores are a key technology for accelerating training and inference algorithms in deep learning. However, they can be used for any algorithm that can be expressed as matrix-matrix multiplications and operates on limited-precision input data. Beamforming is one of them, provided that multiple beams (i.e. directions or positions) are created from the same input streams, and that the weights used to steer the beams are constant for some period of time (they should not be different for every sample in time). Under these conditions, the algorithm is a multiplication of a matrix of sampled data by a matrix of weights (see also Section II). The input data normally come from Analog-to-Digital Converters of which the accuracy is limited to a dozen bits or less, hence there is no benefit from performing the beamforming multiplications in high precision.

The main contributions of this paper are the introduction and performance analysis of the *Tensor-Core Beamformer* (TCBF), a highly efficient beamforming library for multiple application domains. This library is domain independent, and performs complex-valued matrix-matrix multiplications, hiding the complexity of using tensor cores. In addition, we integrated the library into beamforming applications for radio-astronomical use and medical computational ultrasound imaging use. In these applications, the TCBF is up to a factor 10-100 faster than previous GPU-based beamforming implementations, as well as an order of magnitude more energy efficient. This paper describes and evaluates the library, as well as the use in both application domains.

The paper is structured as follows: Section II provides an overview of the background and related work. In Section III, we introduce the TCBF and discuss its key features. Section IV delves into auto-tuning techniques for optimizing performance and energy efficiency, followed by analysis of the optimized TCBF. Next, the applications of the TCBF in radio astronomy and medical ultrasound are detailed in Section V. Finally, we conclude with a summary of findings and future directions in Section VI.

II. Background and related work

Beamforming is a signal-processing technique used to direct the reception or transmission of signals in a specific direction by combining signals from multiple sensors, typically in an array configuration [1]. This process enhances the signal-to-noise ratio (SNR) for the signal of interest while suppressing interference from other directions. Beamforming is for example used to search for pulsars or Fast Radio Bursts [2] in radio astronomy, and for medical ultrasound imaging [3].

Consider an array of N sensors receiving a plane wave signal $s(t)$ from a far-field source at an angle θ. The signal received by the k-th sensor can be expressed as:

$$x_k(t) = s(t - \tau_k) + \sigma_k(t) \tag{1}$$

where τ_k is the time delay associated with the k-th sensor due to the wavefront's angle of arrival, and $\sigma_k(t)$ is the noise at the k-th sensor. The time delay τ_k is given by:

$$\tau_k = \frac{d_k \sin\theta}{c} \tag{2}$$

where d_k is the distance of the k-th sensor from a reference point in the array, and c is the speed of the wave (e.g. speed of light for electromagnetic waves, speed of sound for acoustic waves).

In beamforming, the signals received by each sensor are combined with appropriate weights to form the beamformed output $y(t)$:

$$y(t) = \sum_{k=1}^{K} w_k x_k(t) \tag{3}$$

where w_k are the complex weights applied to each sensor's signal. These weights are designed to steer the beam in the desired direction, effectively aligning the phases of the signals from the desired source and canceling out interference from other directions. When multiple samples are beamformed at once, Eq. 3 maps to a matrix-matrix multiplication. Matrix-matrix multiplication is typically described as the product of an $M{\times}K$ matrix with an $K{\times}N$ matrix. The result is an $M{\times}N$ matrix. In the beamforming algorithm, M corresponds to the number of beams, N is the number of samples beamformed at a time, and K is the number of elements that is summed over, i.e. the number of receivers in the above description.

The beamforming procedure becomes more complicated for near-field sources and when transmissions of an (acoustic) signal are included as well, but the overall procedure remains the same and can still be mapped to a matrix-matrix multiplication.

Tensor cores have been used before for signal processing: The Tensor-Core Correlator [4] is a highly (energy) efficient library that correlates the signals from multiple radio telescopes. Correlating is the "other" method to combine the signals from multiple sensors, and is, for example, used to create sky images. Whereas beamforming is a weighted addition of all sensor signals, correlations are pair-wise multiplications.

Highly-efficient GPU matrix-matrix multiplication libraries already exist, such as CUTLASS and cuBLAS/rocBLAS. However, these libraries typically impose restrictions on data layouts and some lack full support for 1-bit and complex-valued computations. The lower-level Warp Matrix Multiply-Accumulate interface allows precise control over data placement in shared and device memory, enabling a custom kernel optimized for our needs.

Recent AMD GPUs have their own version of tensor cores, called matrix cores. The domain-independent layer of the tensor core beamformer supports AMD matrix cores as well. Although this paper typically uses NVIDIA terminology, the content applies to the AMD equivalent as well. The only exception is 1-bit precision, which is only supported on NVIDIA GPUs.

III. The Tensor-Core Beamformer

The core of the beamforming algorithm is a complex-valued matrix-matrix multiplication, which we have implemented in a separate library, `ccglib`[1]. `ccglib` supports both CUDA and HIP. To program the tensor cores of NVIDIA GPUs, we use the Warp Matrix Multiply Accumulate (WMMA) interface. AMD implements a similar interface through rocWMMA, available in HIP. The user of `ccglib` can switch between the CUDA and HIP backends with a CMake flag, or when compiling manually simply by switching between the *nvcc* and *hipcc* compilers. The use of the tensor cores and the complexity of supporting both HIP and CUDA is hidden from the user. The user only has to provide the input and output matrices and tell `ccglib` what shapes and types the matrices have. To achieve optimal performance, `ccglib` compiles the GPU kernel at runtime with knowledge of both the type of GPU used, and of all input parameters such as the number of receivers and the number of beams to be created. It is also possible to execute several matrix-matrix multiplications at once through a batch size option.

`ccglib` currently supports 16-bit float and 1-bit integer precision. We focus on 16-bit float processing because some domain-specific input data is naturally in this format, making it both practical and efficient. 16-bit tensor-core operations offer significant speedups over 32-bit float on standard GPU cores, while also halving memory usage and bandwidth requirements.

1-bit processing enables higher throughput by reducing memory bandwidth and increasing arithmetic intensity, as the same number of operations are performed on fewer bits. Since 1-bit arithmetic is faster than 16-bit, this can lead to significant speedups. While lower precision introduces quantization noise, beamforming remains robust since many values are accumulated. By leveraging tensor cores for 1-bit arithmetic, we explore its potential for efficient, high-performance beamforming.

In addition to a matrix-matrix multiplication GPU kernel, `ccglib` implements two more types of kernels: For 1-bit precision, the input data must be packed, i.e. 32 consecutive 1-bit samples must be stored in a single 32-bit integer. Packing and unpacking kernels are provided to handle this. Additionally, the matrix-matrix multiplication kernel requires that the input matrices are tiled in device memory. This can be handled by `ccglib` through a transpose kernel. The packing and transpose kernels are relatively straightforward, and both are bound by memory bandwidth as they only move data around.

Beamforming for a specific scientific domain can be implemented as a thin wrapper around `ccglib`. Two such applications are described in Sect. V.

[1] https://git.astron.nl/RD/recruit/ccglib

During the implementation of ccglib, we identified several challenges: the absence of support for complex numbers, the details of 1-bit arithmetic, the limited support for 1-bit tensor-core operations by the NVIDIA Hopper GPU architecture, and the fact that tensor cores have such a high compute throughput that it is difficult to feed them data fast enough. Before discussing these challenges, we investigate the potential of tensor-core technology through a set of micro-benchmarks on a range of workstation and server-grade GPUs: NVIDIA's RTX 4000 Ada (Hereafter AD4000), Tesla A100 (A100), and Grace Hopper (GH200), as well as AMD's Radeon Pro W7700 (W7700), Instinct MI210 (MI210), Instinct MI300X (MI300X), and Instinct MI300A (MI300A).

A. *Tensor-core micro-benchmarks*

Tensor cores support several precisions and matrix sizes, and different GPU architectures support different combinations of these parameters. To get an overview of the attainable tensor core performance, we have run micro-benchmarks on several GPU architectures. These micro-benchmarks do not load data from global memory, to avoid memory throughput bottlenecks (see also Sect. III-C). The benchmarks were run using the cudapeak[2] library. The results are summarized in Table I.

When computing the peak performance using the measured clock frequency, which differs from the theoretical maximum, cudapeak's performance is close to the peak on all GPUs, except for the GH200, which falls notably short. The GH200 and other GPUs of the Hopper generation support a new interface to the tensor cores, called WGMMA. Only with this interface, it is possible to reach maximum performance. As shown in [5], the WMMA interface limits the performance to $60-65\%$ of the maximum. Our benchmark indeed reaches $\sim 65\%$ of the expected GH200 peak performance.

For 1-bit precision, the $16\times8\times256$ matrix fragment layout is not available through the WMMA interface, only through inline PTX. In both cudapeak and ccglib, we have included an extension to WMMA with support for this fragment layout. The 1-bit benchmarks were run with both the WMMA-supported layout of $8\times8\times128$ as well as with this custom extension, and with both XOR and AND as multiplication operand. This leads to a total of four different benchmarks. These are only run on NVIDIA GPUs, as 1-bit matrix values are not supported on AMD GPUs.

The $8\times8\times128$ and $16\times8\times256$ layouts have the same performance on the AD4000, but on the A100 and GH200 the larger layout is at least twice as fast. As the larger layout is never slower than the smaller one, there seems to be no reason to use the small layout when considering just the tensor core throughput. We also note that on the GH200, using XOR as an operand is up to five times slower than using AND. The CUDA documentation notes that XOR is deprecated as of the Hopper generation. However, the instruction is still available at both the WMMA and PTX level. Inspecting the generated SASS assembly reveals that the XOR operation has been removed from hardware, and in software it is replaced by several AND operations combined with boolean logic. This is the reason for the low performance of the XOR mode on the GH200. In the best-performing case, we see the same $\sim 65\%$ of maximum performance as for float16, resulting from our use of the WMMA interface instead of WGMMA.

B. *Complex number support*

Tensor cores were created to accelerate common computations in deep learning and are only capable of executing real-valued matrix-matrix multiplications. Additionally, they only provide an accumulation operation, subtraction is not available. To implement the multiplication of two complex numbers on tensor cores, we need both, though.

Given two complex numbers a and b, complex multiplication is defined as follows:

$$\mathrm{Re}(a\times b)=\mathrm{Re}(a)\,\mathrm{Re}(b)-\mathrm{Im}(a)\,\mathrm{Im}(b)$$

$$\mathrm{Im}(a\times b)=\mathrm{Re}(a)\,\mathrm{Im}(b)+\mathrm{Im}(a)\,\mathrm{Re}(b)$$

Starting with output initialized to zero, this can be implemented for matrices on the tensor cores in five steps:

1) $\mathrm{Re}(a\times b)\mathrel{+}=\mathrm{Re}(a)\,\mathrm{Re}(b)$
2) $\mathrm{Im}(a\times b)\mathrel{+}=\mathrm{Re}(a)\,\mathrm{Im}(b)$
3) $\mathrm{Im}(b)\qquad\quad=-\,\mathrm{Im}(b)$
4) $\mathrm{Re}(a\times b)\mathrel{+}=\mathrm{Im}(a)\,\mathrm{Im}(b)$
5) $\mathrm{Im}(a\times b)\mathrel{+}=\mathrm{Im}(a)\,\mathrm{Re}(b)$

Hence, complex matrix-matrix multiplication can be implemented using four real-valued matrix-matrix multiplications and one negation of the imaginary part of the b matrix. The negation of $\mathrm{Im}(b)$ is executed in local registers, so it is fast and does not modify the global input data.

C. *Need for data reuse*

To achieve good performance on tensor cores, it is of utmost importance to ensure the data are efficiently reused throughout the GPU memory hierarchy, from global memory, L2 and L1 caches, and shared memory, to registers.

The GPU kernels in ccglib are adaptive in the amount of work per thread block and warp, which affects the amount of reuse at the shared-memory level and register-file level, respectively. Optimal configurations for specific GPUs have been determined through auto-tuning as described in Sect. IV-A. ccglib automatically selects these parameters at runtime, the user does not need to provide them. Through this mechanism, we ensure optimal reuse of data at all levels of memory.

In addition to data reuse, we can reduce memory bottlenecks by making use of asynchronous data copies between GPU global and shared memory, available on NVIDIA Ampere and later GPUs. With this feature, it is possible to overlap computations with data transfer to reduce execution time. We implement this by creating a multi-stage buffer in shared memory. While data is being copied to one buffer, another buffer can be copied to the register file and used for computations. Using the CUDA pipeline synchronization primitives, we ensure that data has been written to a shared memory buffer

[2] https://gitlab.com/astron-misc/cudapeak/

TABLE I: Tensor core micro-benchmark results for 16-bit float and 1-bit integer precision. The measured tensor core throughput as well as the theoretical value are shown. 1-bit precision was benchmarked with two matrix fragment layouts and two operands for the multiplication operation. It is available on NVIDIA GPUs only.

Input / output type	Fragment size	Measured performance / Theoretical peak (TOPs/s)						
	$M \times N \times K$	AD4000[a]	A100	GH200	W7700[a]	MI210	MI300X[b]	MI300A[b]
float16 / float32	$16 \times 16 \times 16$	117 / 107	308 / 312	646 / 990	59 / 57	174 / 181	1205 / 1307	949 / 981
int1 / int32 (XOR)	$8 \times 8 \times 128$	1847 / 1710	2465 / 4992	979 / 15800[c]	N/A	N/A	N/A	N/A
int1 / int32 (AND)	$8 \times 8 \times 128$	1804 / 1710	2408 / 4992	3894 / 15800[c]	N/A	N/A	N/A	N/A
int1 / int32 (XOR)	$16 \times 8 \times 256$	1865 / 1710	4942 / 4992	2361 / 15800[c]	N/A	N/A	N/A	N/A
int1 / int32 (AND)	$16 \times 8 \times 256$	1865 / 1710	4942 / 4992	10276 / 15800[c]	N/A	N/A	N/A	N/A

[a] The AD4000 and W7700 run at boosted clock speeds beyond their vendor's specification, explaining why they perform better than the theoretical maximum. [b] The MI300X and MI300A cannot sustain the maximum clock speed in this synthetic benchmark, leading to lower performance than the theoretical value. [c] NVIDIA does not provide the theoretical 1-bit performance for the GH200. We assume it scales from float16 the same as on the Ampere and Ada generation GPUs.

before it is read by the threads and written to the register file. The number of buffers is tunable and automatically set to one on AMD GPUs, which do not support these asynchronous copies.

D. 1-bit arithmetic on tensor cores

In a 1-bit representation of a real number, only two possible values exist. A natural choice is to use these two values to represent -1 and 1, as they preserve sign information. Importantly, this implies that the number 0 cannot be represented.

For 1-bit complex numbers, there is one bit per component, meaning that both the real and imaginary parts are independently represented using a single bit. Preserving sign information along both the real and imaginary axes, this results in four possible complex values. These values are equally spread around the unit circle in the complex plane, as shown in Fig. 1.

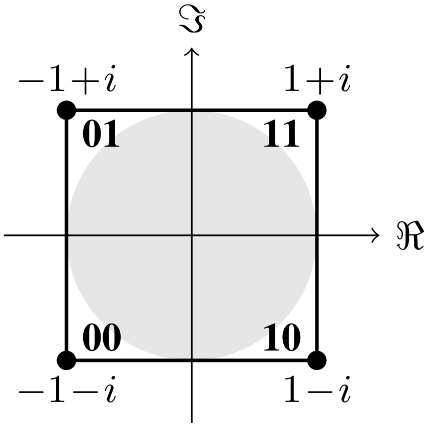


Fig. 1: 1-bit complex numbers and their binary representation. The representable values $-1-i$, $-1+i$, $1-i$, and $1+i$ are shown at the corners of the square, with binary values `00`, `01`, `10`, and `11`, respectively. The light gray circle represents the unit circle. Note that zero, i.e. $0+0i$, is not representable.

NVIDIA tensor cores support 1-bit precision matrix-matrix multiplication using binary operations. Instead of numerical multiplication, they perform a bitwise operation between two input matrices, followed by a population count (popc), which counts the number of bits set to one in the result. The bitwise operation is either XOR (deprecated as of the Hopper architecture) or AND (introduced with the Ampere architecture).

Real-valued matrix-matrix multiplication with the encoding described above can be implemented efficiently using XOR as the bitwise operation. To illustrate this, consider the dot product of two vectors A and B of length K, as shown in Table II for $K = 4$. The left half of the table shows the vector dot product in decimal. After performing element-wise multiplication, the final value of the dot product is obtained by summing the results.

In the binary case, shown in the right half of the table, the process differs slightly. Instead of multiplying the decimal values directly, we perform an element-wise XOR operation between the corresponding elements of the two vectors. This produces a new vector where a binary zero represents a positive value and a binary one represents a negative value. Due to the XOR operation, the binary encoding in this resulting vector is flipped with respect to the binary encoding in the input vector. The final value of the dot product is then determined by subtracting the number of binary zeroes from the number of binary ones in the resulting vector. The number of ones is counted using the population count popc function, while the number of zeroes is simply K minus the number of ones. Thus, the final expression for the 1-bit vector dot product is:

$$K - 2\,\mathrm{popc}(A \oplus B), \tag{4}$$

where $\oplus$ denotes the element-wise XOR operation. Since matrix-matrix multiplication can be expressed as a series of vector dot products, this approach can also be applied to implement matrix-matrix multiplication with 1-bit precision.

TABLE II: Vector dot product in 1-bit precision. Every row in this table corresponds to one element of the input vector, i.e. the input A of length 4 ($K = 4$) has decimal values 1, -1, 1 and -1 and is represented as 1010 in binary.

Decimal			Binary		
A	B	$A_k \times B_k$	A	B	$A_k \oplus B_k$
1	1	1	1	1	0
-1	1	-1	0	1	1
1	-1	-1	1	0	1
-1	-1	1	0	0	0
$\sum A_k \times B_k$		0		$\mathrm{popc}(A \oplus B)$	2
				$K - 2\mathrm{popc}(A \oplus B)$	0

For complex-valued matrix-matrix multiplication, we need to consider two things: the number of matrix multiplications executed and the absence of a representation of zero in the input values.

Firstly, to compute the real and imaginary parts of the output, two separate real-valued matrix-matrix multiplications are required. This means that instead of K terms, $2K$ terms are summed for each part.

Secondly, when multiplying matrices that do not exactly match the sizes supported by the tensor cores, padding is applied to make the matrix dimensions compatible. The padded area is typically set to zero. However, zero cannot be represented in 1-bit mode. Instead, we set the padded region to binary 0, which corresponds to decimal -1. This introduces an additional effect that must be accounted for. For the real part of the output, the results of the two matrix-matrix multiplications cancel out the padding effect, as the results are subtracted from each other. For the imaginary part, however, the padding effect leads to an erroneous addition of $K_{\mathrm{pad}} \times -1 \times -1$ in the result, where K_{pad} denotes the amount of padding. Combining these two effects, we arrive at the following two equations for complex-valued 1-bit matrix-matrix multiplication on tensor cores, using subscript r and i to denote the real and imaginary parts:

$$(A \times B)_{\mathrm{r}} = 2\Big(K - \big(\mathrm{popc}(A_{\mathrm{r}} \oplus B_{\mathrm{r}}) + \mathrm{popc}(A_{\mathrm{i}} \oplus \overline{B}_{\mathrm{i}})\big)\Big) \quad (5)$$

$$(A \times B)_{\mathrm{i}} = 2\Big(K - K_{\mathrm{pad}} - \big(\mathrm{popc}(A_{\mathrm{r}} \oplus B_{\mathrm{i}}) + \mathrm{popc}(A_{\mathrm{i}} \oplus B_{\mathrm{r}})\big)\Big)$$

E. NVIDIA Hopper support

As explained in the previous section, 1-bit matrix-matrix multiplication can be efficiently implemented with an XOR operation. However, this operation is deprecated as of the Hopper architecture and leads to low performance, as discussed in Sect. III-A and shown in Table I.

To optimize performance on Hopper, we switch to using the AND operation. The XOR operation detects when two input bits are different: if they are different, the output is set to 1; otherwise, it is set to 0. We can achieve similar functionality with the AND operation by following a different sequence of steps. Specifically, we perform an AND operation on the inputs, followed by negating both inputs, then performing another AND operation, and finally summing the results of both AND operations. This method detects when the input bits are the same, as opposed to when they are different.

This means the (signed) output of the matrix-matrix multiplication is negated relative to the XOR version. In summary, (real-valued) 1-bit matrix-matrix multiplication can be implemented with the AND instruction as follows:

$$2\big(\mathrm{popc}(A \wedge B) + \mathrm{popc}(\overline{A} \wedge \overline{B})\big) - K, \qquad (6)$$

where $\wedge$ denotes the element-wise AND operation. Although using the AND operation requires twice as many tensor core instructions compared to XOR, this still results in a net performance improvement on Hopper because the AND operation is up to five times faster than XOR on this architecture. Therefore, `ccglib` automatically switches to the AND-based matrix-matrix multiplication when a Hopper or newer NVIDIA GPU is detected.

IV. Performance and energy efficiency

A. Auto-tuning

GPU kernels can typically be run on a GPU with many different thread block dimensions, that all give the correct result but can result in vastly different performance. Additionally, the GPU kernels in `ccglib` were designed such that parameters, like the amount of work per thread block and warp, can be set at compile time. Because `ccglib` compiles the GPU kernels when the application is running on the host, it can pick different values for these tunable parameters based on the type of GPU used as well as the input data sizes. To find the optimum of the tunable parameters, we need to explore a vast search space, and this process has to be repeated for each GPU architecture. To facilitate this, we use Kernel Tuner [6], a Python-based auto-tuning framework that can automatically optimize kernels written in both CUDA and HIP [7]. Kernel Tuner measures the run time of each configuration of a GPU kernel. It is possible to extend Kernel Tuner with other metrics, either built-in or custom. In addition to performance metrics, we measure the energy consumption of the GPU using the `Power Measurement Toolkit (PMT)` [8]. `PMT` supports power measurements of both NVIDIA GPUs through `NVML`, as well as AMD GPUs through `rocm-smi`.

In `ccglib` we have three types of GPU kernels: a packing kernel for 1-bit data, a transpose kernel, and matrix-matrix-multiplication kernels for 16-bit float and 1-bit integer types. Only the matrix-matrix multiplication kernel is always invoked. The use of the others depends on the earlier steps of the processing pipeline `ccglib` is used in. Additionally, the matrix-matrix multiplication kernels take most time and are the only kernels that have tunable parameters other than the thread block size. Hence, we focus our optimization process on these kernels.

The optimal tuning parameters do not only depend on the GPU, but also on the size of the input and output data as well as the precision used. As a generic use case, we tune the float16 kernel for $M = N = K = 8192$, while for 1-bit integer we select $M = 32768$, $N = 8192$, and $K = 524288$. To assess a kernel, we define the performance in TOPs/s as the number of useful operations, i.e. $8 \times M \times N \times K$, per second. In the limit of large matrices, the product of the matrix sizes is the number of fused multiply-add (FMA) instructions required for real-valued matrix-matrix multiplication. The factor eights comes from the fact that four FMA instructions are required for each complex multiplication, and each FMA counts as two instructions. The resulting performance number is divided by the average power consumption of the GPU during the kernel execution to obtain the number of operations per second per Watt, or equivalently the number of operations per Joule.

The performance and energy efficiency of each combination of tuning parameters is shown in Fig. 2. Typically, the most performant combination of parameters is also the most energy efficient solution. On the GH200, there is a large spread of kernels with similar performance, but up to a factor two difference in energy efficiency.

A summary of the parameters for the fastest kernels is given in Table III. In float16, the MI300X is both the fastest and most energy-efficient GPU. The GH200 is the fastest in int1, although the A100 is more energy efficient. The optimal tuning parameter values typically vary a lot from GPU to GPU. The MI300X and MI300A optimal parameters are identical, which is not surprising given that they are built using identical architectures but with a different number of accelerator complex dies. While a default set of parameters is shipped with `ccglib`, a GPU-specific optimization is best.

B. Roofline analysis

After auto-tuning, we know that we reach the maximum performance obtainable for our implementation of the matrix-matrix multiplication algorithm. However, we also want to assess whether or not we reach good performance relative to each GPU's capabilities. In addition to the maximum tensor core throughput discussed in Sect. III-A, we need to consider the memory throughput. This naturally leads to a roofline analysis, where we compare our implementation to the theoretical maximum obtainable on each GPU. To construct the ceiling of the roofline, we use the theoretical memory bandwidth of the GPU and the measured peak tensor core throughput (see Table I). For both the 16-bit and 1-bit kernels, we then select a small and large matrix size and tune the kernel parameters as described in Sect. IV-A, and select the best-performing kernel. The matrix sizes are set as follows $(\text{batch size} \times M \times N \times K)$: float16 small - 256×1024×1024×64, float16 big - 1×8192×8192×8192, int1 small - 256×1024×1024×256, int1 big - 1×32768×8192×524288. The performance and number of operations are defined the same way as during the tuning. We then use the theoretical amount of bytes transferred to and from device memory to calculate the arithmetic intensity (AI).

The resulting rooflines are shown in Fig. 3. For all GPUs, the small matrix size is memory-bound. On all GPUs, but especially the NVIDIA GPUs, we reach a performance very close to the limit set by the memory bandwidth. The larger matrix size is compute bound, and reaches $50 - 85\%$ of the peak tensor-core throughput. In all cases except the small matrix size on the workstation-grade GPUs, `ccglib` is faster than the theoretical maximum of the normal single-precision cores by a wide margin.

C. Benchmarking

We have shown that we reach good performance on a specific set of matrix sizes, however we aim for a solution that is generally applicable and reaches good performance for a wide range of matrix sizes. While it is possible to auto-tune the matrix-matrix multiplication kernel for each potential matrix input size, this is not feasible in practice. Instead, we take the best parameters from Table III, and use the built-in benchmark tools of `ccglib` to measure performance and energy efficiency across a range of matrix sizes.

The results are shown in Fig. 4. For all GPUs and precisions, the performance and energy efficiency is substantially lower for smaller matrices. However, starting from matrices of a few thousand elements on each side, we typically reach close to optimal performance. The performance is best when the matrix size is a multiple of the amount of work per thread block. Otherwise, data are padded and the performance is relatively lower. This is the cause of the sawtooth pattern in the results. Overall, `ccglib` performs well on a large range of matrix sizes.

V. Applications

A. Computational ultrasound imaging

Computational ultrasound imaging (cUSi) is a recent advancement in the field of medical ultrasound. It allows for 3D imaging using a spatially under-sampled transceiver array in conjunction with an spatial encoding mask and a large computational model to decode the spatial information needed to form an image [9]. The cUSi technique essentially changes the sensing problem to a compute problem. This year, Brown et al. showed that using cUSi it is possible to obtain 3D images of blood flow in a mouse brain using an array of only 64 transceivers, which normally requires several thousands of transceivers [10].

However, the caveat of this technique is the number of computations needed to form an image, which makes it currently not possible to obtain real-time imaging feedback. The imaging reconstruction relies on the multiplication of a measurement matrix with an acoustic model matrix which contains for every voxel in the image volume (number of columns) all the expected pulse-echo signals for each transceiver and for each measurement (number of rows). Typically, the minimum number of voxels is 128^3 and the number of rows for a 64-transceiver probe is 128 (temporal frequencies) $\times$ 64 (transceivers) $\times$ 32 transmissions. The measurement matrix has the same number of rows as the model matrix and the number of columns equals the number of repeated measurements from which, in the case of imaging blood flow, the Doppler signal is computed. This number, which is named ensemble size, can range from 100-10000 frames. In this example case we use $\sim$ 8000 frames.

The real-time constraints for this problem are also challenging. Considering a pulse-echo repetition frequency of 32 kHz and an ensemble size of 8000, the time required for the image reconstruction (matrix-matrix multiplication) should be less than 8 seconds in order to maintain real-time feedback.

In this work we show the use of an ultrasound tensor-core beamformer implemented as a wrapper around `ccglib`. We tackle the real-time feedback problem by shrinking the volume size to either a smaller sub-volume, as we do in this example case, or several orthogonal planes through the volume. In addition, we explore a further reduction of the required memory by only keeping the sign of the signal both

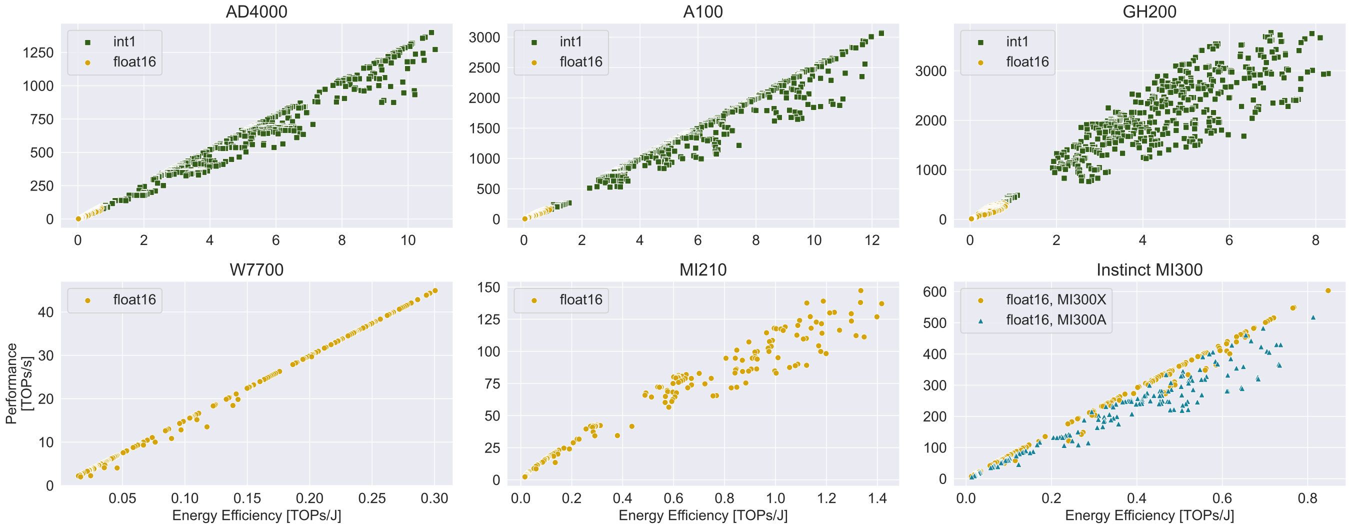


Fig. 2: Auto-tuning results of `ccglib` matrix-matrix multiplication kernel. The measured performance and energy efficiency of each combination of tuning parameters is shown.

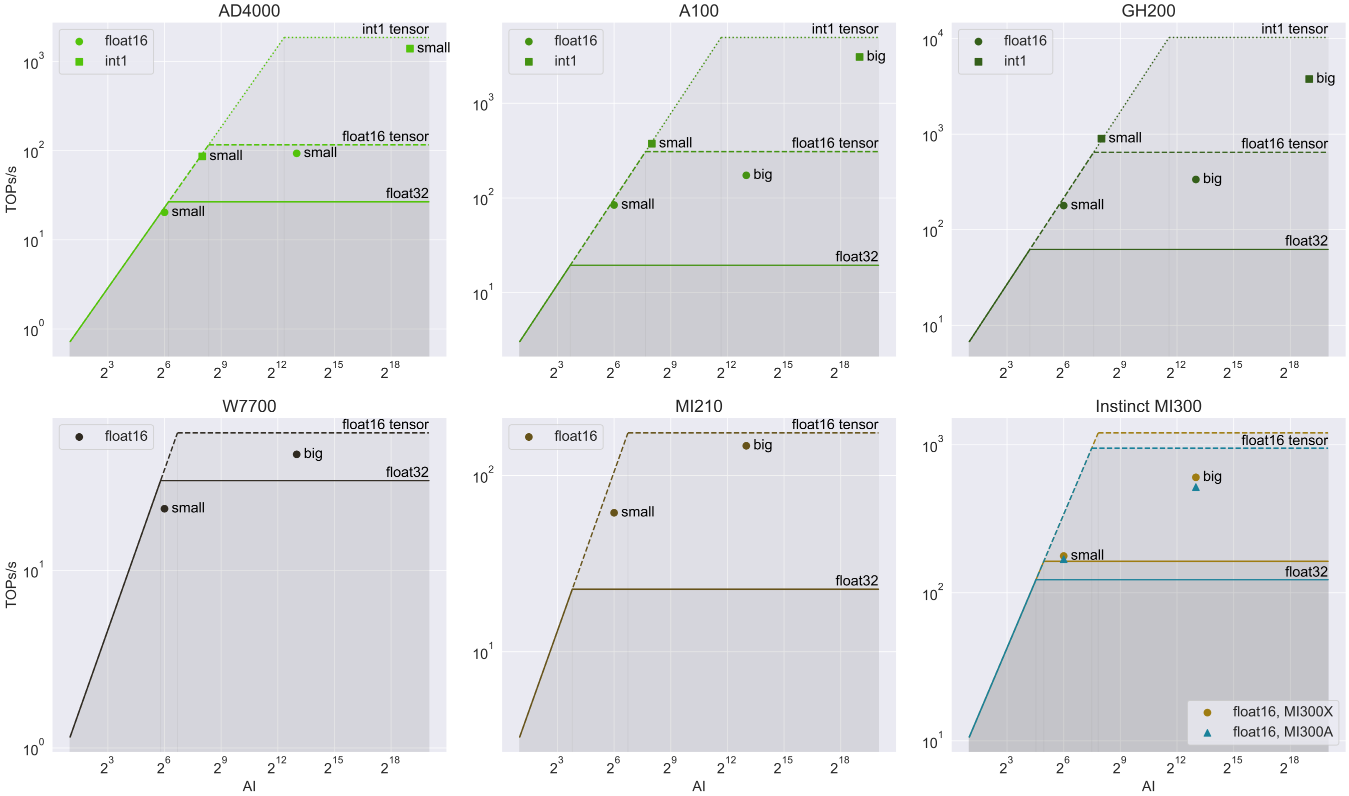


Fig. 3: Roofline analysis of the `ccglib` matrix-matrix multiplication kernel. For each GPU, we show the roofline ceiling of the float16 and int1 (NVIDIA only) tensor cores, as well as the normal float32 cores for comparison.

TABLE III: Matrix-matrix multiplication kernel performance, energy efficiency, and optimal tuning parameter values.

GPU	Precision	TOPs/s	TOPs/J	M per block	M per warp	N per block	N per warp	Number of buffers
AD4000	float16	93	0.7	256	32	32	32	2
A100	float16	173	0.8	256	64	32	32	2
GH200	float16	335	0.8	128	64	64	32	2
W7700	float16	45	0.3	256	128	64	16	1
MI210	float16	147	1.3	128	64	64	32	1
MI300X	float16	603	0.9	128	64	128	32	1
MI300A	float16	518	0.8	128	64	128	32	1
AD4000	int1	1400	10.7	256	128	32	16	2
A100	int1	3080	12.3	128	32	64	64	4
GH200	int1	3780[a]	6.0[a]	64	64	128	32	2

[a] This performance number is with respect to the theoretical amount of useful operations. Because the GH200 uses AND-based tensor-core instructions (see Sect. III-E), which require twice the number of instructions, the actual throughput of the tensor cores is twice as high.

in the measurement matrix as well as the model matrix. In this approach the data only requires single bit precision. Note that the Doppler processing is done before extracting the sign. Otherwise, the Doppler signal will be lost in the dominant stationary signals.

In Fig. 5 we show the number of frames per second that the TCBF can sustain on different GPUs. The processing includes the 1-bit packing and transpose of the measurement matrix. It excludes these steps for the model matrix, as this typically happens once before the experiment and does not need to be repeated. For a set of three orthogonal planes, all three GPUs can easily sustain the required real-time frame rate of 1000 frames per second. None of the GPUs can process the full 128^3 data volume in real time, although the GH200 is capable of processing $\sim 85\%$ of the voxels in real time. Reducing for example the number of frequencies from 128 to 64 would make real-time processing of the full data volume possible for both the A100 and GH200.

In addition to the real-time system, we explore the use of the TCBF for beamforming of pre-recorded data. In this case, there is no real-time constraint. However, quick feedback on experimental results is still important. As a dataset we use the anesthetized mouse brain dataset presented in [10]. We beamform a subset of the volume, with a total of $36 \times 30 \times 30$ voxels. The dataset contains 8041 frames, each with 128 temporal frequencies, 64 transmissions, and 64 transceivers. This leads to a matrix-matrix multiplication with shape $M = 38880$, $N = 8041$, $K = 524288$. Excluding reading the data from disk, the TCBF can process these data in $1.2\,\mathrm{s}$, which significantly shorter than the real-time requirement of $8\,\mathrm{s}$, leaving room for e.g. Doppler processing. Ultrasound processing is typically done in Matlab, Python or Octave. As a comparison, we run the matrix-matrix multiplication in float32 precision using Octave with OpenCL backend. On an A100, this takes roughly 15 minutes. The TCBF is nearly three orders of magnitude faster, allowing, for the first time, real-time feedback on such large volumes. While conversion to 1-bit means that the contrast is reduced, combining this much data still results in usable image feedback as shown in Fig. 6.

B. Radio astronomy

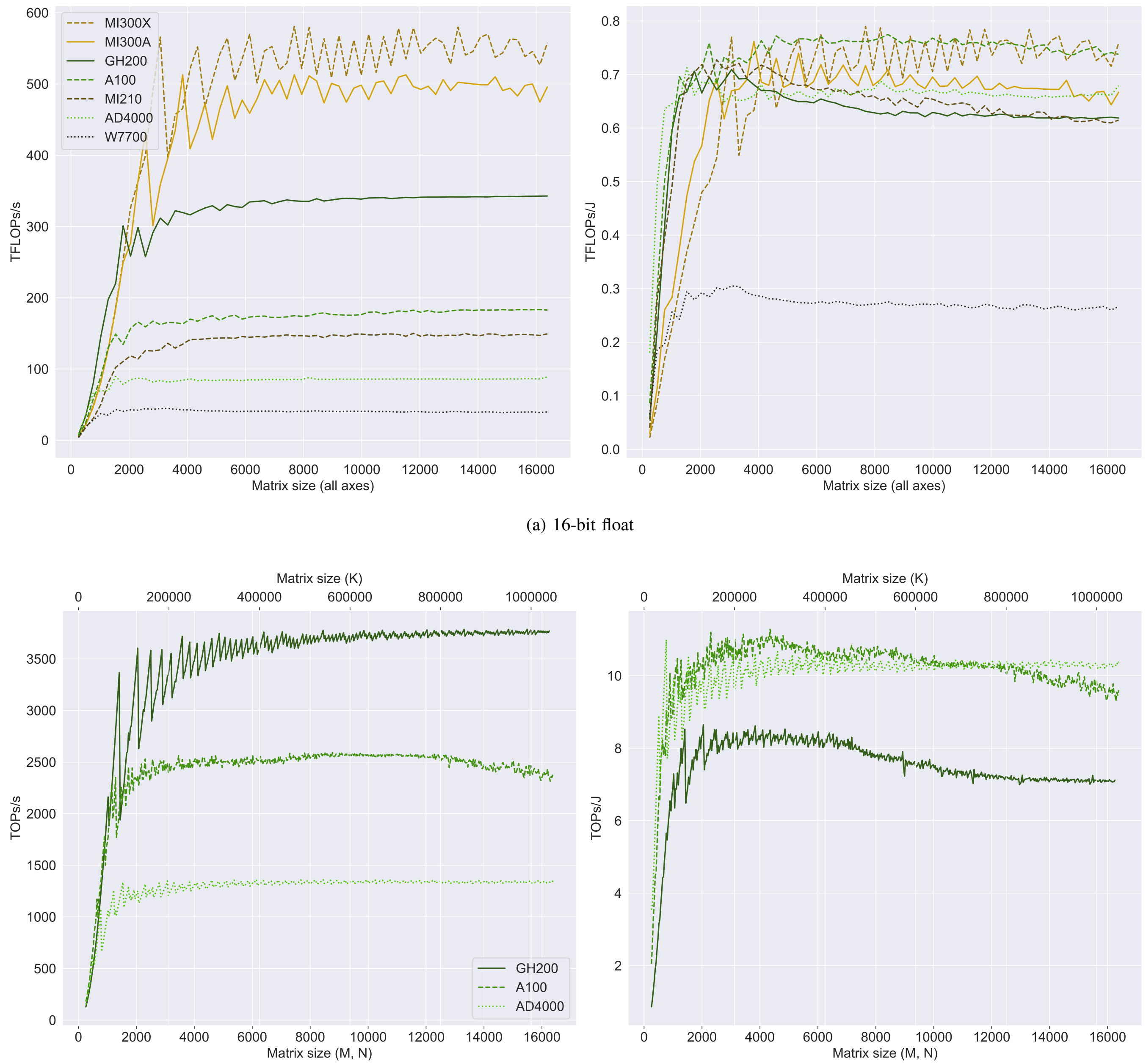
In radio astronomy, beamforming techniques are employed to enhance signal detection by combining data from multiple antennas.

LOFAR (Low-Frequency Array) [11] is a radio telescope network consisting of tens of geographically distributed stations across Europe. Each station is composed of numerous individual antennas that collectively capture radio signals from the sky. These signals are initially processed by a station beamformer, implemented on Field-Programmable Gate Arrays (FPGAs) within each station. The station beamformer combines the signals from all antennas in the station into a coherent station beam, effectively pointing the station at a specific region of the sky. The resulting data, known as beamlet data, is then transmitted to a central beamformer [12], where the signals from all stations are coherently combined. This central processing step allows LOFAR to achieve high sensitivity and resolution, enabling detailed observations of astronomical phenomena across a wide field of view.

The central processing facility employs a second stage of beamforming, which can perform either coherent or incoherent beamforming. Coherent beamforming preserves phase information by aligning the signals from each station, producing a high-resolution, narrow beam with increased sensitivity. This approach is computationally intensive but is essential for high-angular-resolution observations, such as pulsar studies or imaging of compact objects. In contrast, incoherent beamforming discards phase information and instead combines the power from each station, creating a broader beam with a wider field of view but lower resolution. This method is computationally less demanding and is well-suited for all-sky surveys and transient detection.

When observing pulsars and fast transients with an interferometer, achieving high-time resolution is crucial. Typically, a time resolution of $\lesssim 1\,\mathrm{ms}$ is used. To achieve such a high time resolution and retain manageable data rates, the spatial resolution is reduced.

This beamforming approach offers several advantages. Higher angular resolution enhances precise localization and background rejection. Additionally, each station's wide field of view enables high survey speeds, particularly when the entire field can be processed. Multi-beaming capabilities further enhance interference rejection and allow for the construction of a larger total collecting area. However, these benefits come with trade-offs, including a restricted field of view unless multiple beams are synthesized, potentially higher data rates due to



(a) 16-bit float

(b) 1-bit int

Fig. 4: Complex matrix-matrix multiplication benchmark results for (a) 16-bit data and (b) 1-bit data. The left panels show performance, while the right panels show energy efficiency.

numerous data streams, and the need for precise calibration to “phase up” the array. Moreover, this technique can result in a complex instantaneous sidelobe pattern.

LOFAR beamforming is mapped to matrix-matrix multiplication as follows: M represents the number of beams, with each beam corresponding to a row in the resulting matrix. The parameter N is the number of samples (in time), representing the number of columns in the output matrix. K corresponds to the number of stations, reflecting the number of inputs combined during the matrix-matrix multiplication process. Finally, the product of the number of polarizations and channels is the batch size.

A LOFAR tensor-core beamformer is implemented using the 16-bit mode of `ccglib`. As data are typically already GPU-resident and remain on the GPU for further computations, we only consider the matrix-matrix multiplication component for performance analysis. The chosen parameters are 1024 beams, 1024 samples, a range from 8 to 512 stations to be combined, and a batch size of 256. This configuration is also run using the reference LOFAR beamformer on an A100 GPU. It runs

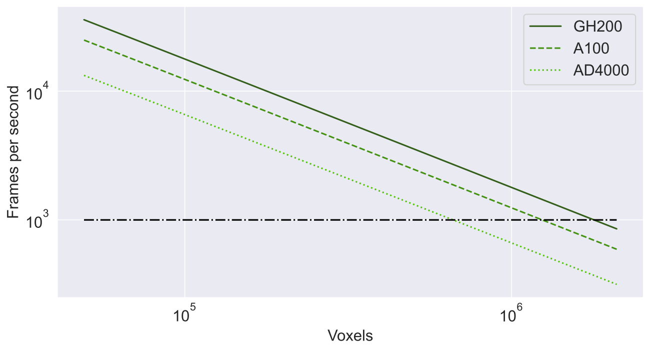


Fig. 5: Performance of beamforming for ultrasound. The number of voxels ranges from three orthogonal planes of 128×128 each, to the full 128^3 data volume. The horizontal dash-dotted line indicates the minimum number of frames per second required for real-time performance.

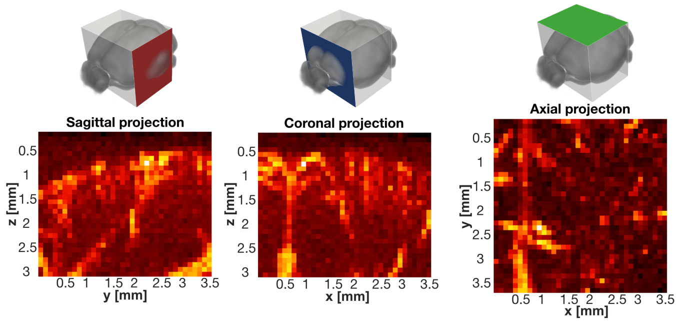


Fig. 6: Three orthogonal (sagittal, coronal and axial) maximum intensity projections through the beamformed volume. The volume which contains the blood flow inside a mouse brain was obtained by averaging the magnitude of the complex beamformed signal along the 8041 frames. See also [10].

in float32 precision on the normal GPU cores. Note that we have removed the calculation of beamformer weights from the reference beamformer, to be able to fairly compare the reference and tensor-core implementations.

The performance and energy efficiency of the LOFAR TCBF are shown in Fig. 7. The sawtooth pattern stems from padding that happens when the number of receivers is not a multiple of the amount of work per GPU thread block set during the auto-tuning of the kernel. Except for very small numbers of receivers, the TCBF outperforms the reference beamformer on both the A100 and GH200, in both throughput and energy efficiency. On the A100, the TCBF is up to 20 times faster and 10 times more energy efficient than the reference beamformer. For the typical LOFAR configuration of 48 stations, the TCBF is still several times faster. The MI300X outperforms the GH200 on this application, achieving up to 50% higher performance, with similar energy efficiency. However, with 512 receivers, the workload is still too small to fully saturate this GPU, preventing it from reaching its peak theoretical performance (approximately twice that of the GH200).

VI. Conclusions and Future Work

We have introduced the Tensor-Core Beamformer, with at its core a high-performance and energy-efficient complex matrix-matrix multiplication library with outstanding performance on both NVIDIA and AMD GPUs. We have shown applications in both medical ultrasound and radio astronomy, where the TCBF improves significantly upon earlier beamformers in both performance and energy efficiency. In medical ultrasound, real-time imaging is essential, allowing a surgeon to change their course of action based on the ultrasound images. Because the TCBF is up to three orders of magnitude faster than previous implementations, this real-time feedback is now for the first time possible for 3D computational ultrasound imaging. The radio-astronomical TCBF is 2-20 times faster than the existing beamformer, as well as 10 times more energy efficient. This makes it possible to either form more beams in real-time, or reduce the amount of hardware needed for beamforming, reducing energy consumption significantly as well.

Several improvements and extensions to `ccglib` are being considered for future releases: Firstly, the tensor cores support more precisions than just float16 and int1. Both NVIDIA and AMD (starting with CDNA3) support tensorfloat32, a 19-bit format with the same range as float32 but less precision. AMD supports float32 as well. Support for these formats is currently available as an experimental feature in `ccglib`. The most recent architectures have introduced several 8-bit float formats, which may become relevant in the future.

Secondly, the matrix-matrix multiplication kernels in `ccglib` currently require a transpose of the input data because the complex data have to be separated into their real and imaginary components, instead of the more usual interleaved storage format. In the future, we would like to provide a matrix-matrix multiplication kernel that does not require this transpose, and works on interleaved real and imaginary data instead. Such a method has already been used successfully in the tensor core correlator [4].

Lastly, for NVIDIA's Hopper and Blackwell generations, new interfaces were introduced for the tensor cores, along with enhancements like the tensor memory accelerator. To achieve maximum tensor core performance, these features must be leveraged. Supporting this in `ccglib` is highly non-trivial, but will likely be important to maximize performance on future GPU generations.

Acknowledgments

This work was supported by the Netherlands eScience Center under grant number ETEC.2020.025 (RECRUIT), by the RADIOBLOCKS grant (HORIZON-INFRA-2022-TECH-01, Grant Agreement nr. 101093934), and by the Netherlands Organization for Scientific Research (NWO) through the DAS-6 grant [13]. This work has made use of resources and expertise provided by SURF Experimental Technologies Platform (SURF-ETP), which is part of the SURF cooperative in the Netherlands, under project no. SURF-ETP0012. We would like to thank AMD for providing access to MI300 GPUs and NVIDIA for donating an A100 GPU.

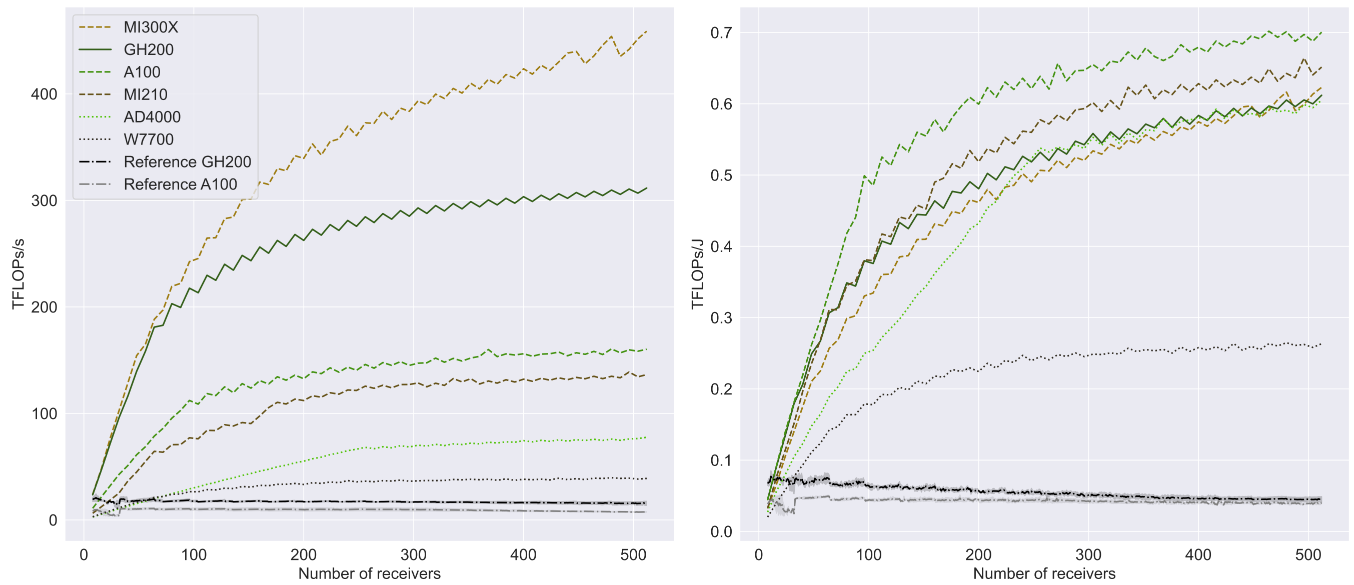


Fig. 7: Performance (left) and energy efficiency (right) of the LOFAR TCBF. The reference lines for A100 and GH200 correspond to the current LOFAR beamformer kernel (without Tensor Cores) running in float32 precision.

References

[1] B. V. Veen and K. Buckley, "Beamforming: a versatile approach to spatial filtering," *IEEE ASSP Magazine*, vol. 5, pp. 4–24, 4 1988. [Online]. Available: http://ieeexplore.ieee.org/document/665/

[2] J. van Leeuwen *et al.*, "The Apertif Radio Transient System (ARTS): Design, commissioning, data release, and detection of the first five fast radio bursts," A&A, vol. 672, p. A117, Apr. 2023.

[3] V. Perrot, M. Polichetti, F. Varray, and D. Garcia, "So you think you can DAS? A viewpoint on delay-and-sum beamforming," *Ultrasonics*, vol. 111, p. 106309, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0041624X20302444

[4] J. W. Romein, "The Tensor-Core Correlator," A&A, vol. 656, p. A52, Dec. 2021.

[5] W. Luo, R. Fan, Z. Li, D. Du, Q. Wang, and X. Chu, "Benchmarking and Dissecting the Nvidia Hopper GPU Architecture," in *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2024, pp. 656–667. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/IPDPS57955.2024.00064

[6] B. van Werkhoven, "Kernel Tuner: A search-optimizing GPU code auto-tuner," *Future Generation Computer Systems*, vol. 90, pp. 347–358, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X18313359

[7] M. Lurati, S. Heldens, A. Sclocco, and B. van Werkhoven, "Bringing Auto-Tuning to HIP: Analysis of Tuning Impact and Difficulty on AMD and Nvidia GPUs," in *Euro-Par 2024: Parallel Processing*, J. Carretero, S. Shende, J. Garcia-Blas, I. Brandic, K. Olcoz, and M. Schreiber, Eds. Cham: Springer Nature Switzerland, 2024, pp. 91–106.

[8] S. Corda, B. Veenboer, and E. Tolley, "PMT: Power Measurement Toolkit," in *2022 IEEE/ACM International Workshop on HPC User Support Tools (HUST)*, 2022, pp. 44–47.

[9] P. Kruizinga *et al.*, "Compressive 3D ultrasound imaging using a single sensor," *Science Advances*, vol. 3, no. 12, p. e1701423, 2017. [Online]. Available: https://www.science.org/doi/abs/10.1126/sciadv.1701423

[10] M. D. Brown *et al.*, "Four-dimensional computational ultrasound imaging of brain hemodynamics," *Science Advances*, vol. 10, no. 3, p. eadk7957, 2024. [Online]. Available: https://www.science.org/doi/abs/10.1126/sciadv.adk7957

[11] M. P. van Haarlem *et al.*, "LOFAR: The LOw-Frequency ARray," A&A, vol. 556, p. A2, Aug. 2013.

[12] P. C. Broekema *et al.*, "Cobalt: A GPU-based correlator and beamformer for LOFAR," *Astronomy and Computing*, vol. 23, pp. 180–192, 4 2018. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S2213133717301439

[13] H. Bal *et al.*, "A Medium-Scale Distributed System for Computer Science Research: Infrastructure for the Long Term," *Computer*, vol. 49, no. 5, pp. 54–63, 2016.