Porting a ray tracing method for determining the target echo strength (TES) of underwater objects to GPUs - a field report

Ralf Burgschweiger[1], Ingo Schäfer[2], Delf Sachau[1] and Jan Ehrlich[2]

[1] Helmut Schmidt University, University of the Federal Armed Forces Hamburg, Faculty of Mechanical Engineering, Chair of Mechatronics, Hamburg Germany
[2] Bundeswehr Technical Center for Ships and Naval Weapons, Maritime Technology and Research (WTD 71), Acoustic Modelling, Eckernförde, Germany

Ralf Burgschweiger, Helmut-Schmidt-Universität / Universität der Bundeswehr Hamburg
Fakultät für Maschinenbau, Institut für Mechatronik
Holstenhofweg 85, 22043 Hamburg, Germany
Phone: ++49-40-65 41-44 94
Email: burgschr@hsu-hh.de

Abstract: *Investigations of a test implementation of the "simple" Kirchhoff approximation method (KIA) for GPU graphics adapters (using OpenCL) have shown that, depending on the GPU hardware available, significant performance improvements can be achieved. Therefore, as part of the "Computational Acoustics" research project, the BEAM ray tracing method, originally implemented in C++, was ported to address the required acoustic problems (bistatic, monostatic calculations and their combination). This article discusses the challenges encountered during this process, shares the practical experiences implementing the complex parallelized algorithm in OpenCL and shows the results achieved. It also highlights the speed advantages compared to conventional high-performance PC workstations using examples of underwater objects.*

Keywords: *Target echo strength, ray tracing, GPU*

1. INTRODUCTION

To estimate the backscattering behaviour of objects under water, the so-called target echo strength (TES) is used, which results from the ratio of incident to reflected sound intensity calculated back to a distance of one meter from the object. Various numerical methods (boundary and finite element methods, approximate methods, etc.) are available for this purpose, which were implemented in high-performance applications as part of the joined research project "Computational Acoustics" at the Helmut-Schmidt-University, Hamburg, and the Bundeswehr Technical Center for Ships and Naval Weapons, Maritime Technology and Research (WTD 71), Eckernförde, Germany.

In recent years, the ray tracing-based solver BEAM [1] has been developed, which is able to calculate the TES of underwater structures consisting of surfaces with appropriate boundary conditions and combined with fluid fillings. Due to its high computational speed, it is also very well suited for sweeps over a given frequency range.

In many practical applications for calculations (image processing, artificial intelligence, games, etc.), the use of graphics processors (GPUs) has proven to be very powerful. Therefore, a sub-task of the project is to implement the BEAM algorithm on GPUs and to determine which performance improvements can be achieved compared to the current "standard CPU code".

This article describes the challenges and practical experience of porting the BEAM method from standard C++ code to OpenCL, gives an overview of the advantages and disadvantages and presents the defining characteristic of a GPU for performance.

The results achieved, also for different hardware, are given using various problem types.

2. BASICS OF THE METHODS / RELEVANT DIFFERENCES

Two years ago at UACE 2023, the advantages of using AVX2-CPU instructions were demonstrated using the approximate Kirchhoff method [2] as an example, for full details see [3]. In the meantime, further work has been done on this and a migration of the KIA process to OpenCL has been implemented for test purposes.

Table 1 shows the new solution times (including the GPU version) for the calculation example at that time (BeTSSi hull, consisting of 178,000 elements, TES values calculated on a spherical section of 0°...180°, elevation ±20°, step width 0.2°, 181,101 evaluation points, averaging over 11 frequencies).

method / variation	**solver time [s]**
BEM	≈ 300,000
MKIA (AVX2)	≈ 31
MKIA (AVX512)	≈ 11
MKIA (GPU, RTX4090)	**2.9**

Table 1: MKIA solving times

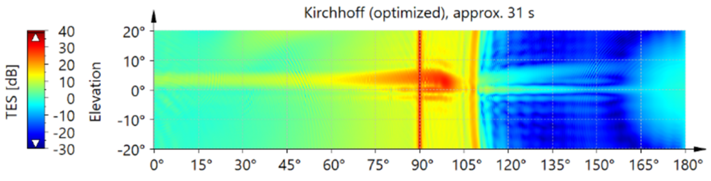


Fig. 1: TES MKIA example (181,101 eval. Points)

Due to the performance advantage of the GPU variant by a factor of 3.8, it was decided to implement an adaptation of the BEAM ray tracing code for GPUs.

2.1. Differences between the methods

The following list provides a brief overview of the differences between the "simple" Kirchhoff method and the "complex" BEAM algorithm in terms of application possibilities and resulting workload that are relevant to this article.

Kirchhoff approximation		**BEAM ray tracer**
● supports only convex surfaces	⇔	● supports convex and concave surfaces
● supports only hulls with predefined reflection factors	⇔	● supports internal structures with mixed boundary conditions
● the computational load depends only on the number of surface elements (one integration per element)	⇔	● the computational load depends on the maximum number of traced child beams per tree (one tree per start beam, see 2.2) - The number of summations doubles per level, i.e., for a maximum tracing level of $N_{b,max} = 7$, up to 255 beams must be considered - A small change in the direction of sound incidence may lead to completely different beam paths

2.2. Ray tracing algorithm

Fig. 2 and 3 show an example of the structural setup of a “beam tree” resulting from ray tracing up to a tracking level of 5 (in 2D, using a spherical shell with an internal rigid angled plate) to illustrate the difference to the linear load distribution in the KIA algorithm.

Starting from a given sound source and direction, a start beam (with level 0) is generated and tracked further by testing for hits on the structure(s).

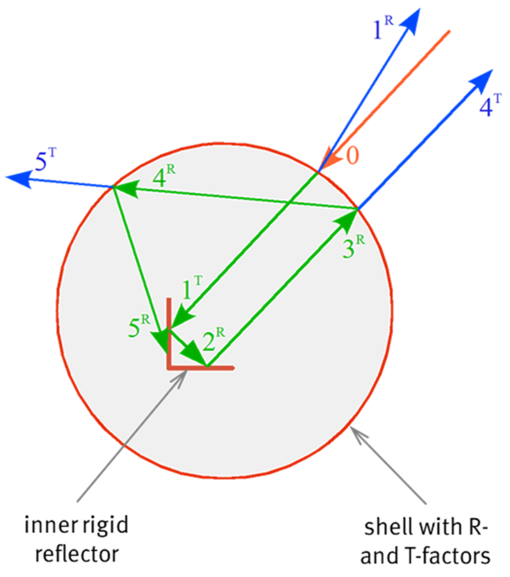


Fig. 2: 2D ray tracing, stop level 5

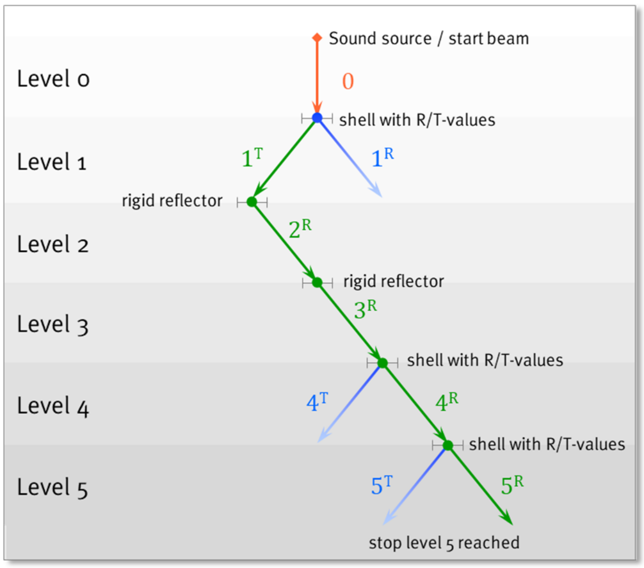


Fig. 3: resulting “beam tree”

Each colored dot ● in fig. 3 corresponds to the “hit” of an element of the surfaces. It is easy to see that, depending on the boundary conditions given, each hit of a beam leads to a decision as to whether a transmitted (#^{T}) and/or reflected child beam (#^{R}) is generated and whether this (or these) must be tracked further.

The BEAM method can be parallelized well when subsets of all required "start beams” are assigned to the available CPU (or GPU) cores, since the tracing of one single beam is independent of all others.

After the complete construction of a beam tree, the complex pressure components per partial beam are added up with regard to the evaluation points in a post-processing calculation.

2.3. Relevant hardware differences

To illustrate the challenges involved in implementing the algorithm for a GPU, the relevant differences between a CPU and a GPU are briefly listed below.

“conventional” CPU		**Graphics processor (GPU)**
● consists of independent cores (up to 192 physical cores per CPU chip or 384 logical cores with hyperthreading)	⇔	● consists of so-called stream processors (**SPs**), organized in compute unit groups (**CUs**), up to 21,760 SPs / 170 CUs per GPU (≙ 128 SPs per CU)
● large and fast cache memory (up to 80 KB L1- and 2 MB L2-cache per core)	⇔	● high speed access to very limited private storage per CU (up to 128 KB)
● fast access to the entire host main memory	⇔	● “slower” access to GPU global memory and need to copy relevant data between host and GPU
● independent instruction execution per core	⇔	● synchronized instruction execution per compute unit (SIMD, single instruction multiple data)

The consequences of these differences for performance will be discussed in more detail later. Table 2 provides an overview of the available and used GPUs:

GPU model	**key data**					**benchmark**	**price**
	CUs	SPs	GPU family	RAM	year	OpenCL	GPU
nVidia GeForce RTX 4090	128	16,384	AD102	24 GB GDDR6x	2022	317,945	1,950 €
nVidia GeForce RTX 4070	56	7,168	AD104	12 GB GDDR6x	2024	222,849	1,000 €
AMD Radeon RX 7900	96	6,144	Navi 31	24 GB GDDR6	2022	212,200	980 €

Table 2: available and tested GPUs

Table 3 provides an overview of the used “conventional” workstations:

CPU model	**key data**						**price**
	cores	threads	Gen.	RAM	year	Core freq.	CPU
AMD ThreadRipper Pro 7975	32	64	Zen 4	256 GB DDR5	10/2023	4.0/5.3 GHz	5,000 €
AMD EPYC 9655	96	192	Zen 5	768 GB DDR5	10/2024	2.6/4.5 GHz	7,000 €

Table 3: available workstations

3. OPENCL

OpenCL (Open Computing Language, [4]) is a framework for writing programs that execute across heterogeneous platforms consisting of central processing units, graphics processing units, digital signal processors (DSPs), field-programmable gate arrays (FPGAs) and other processors or hardware accelerators [5].

OpenCL was chosen to implement the adaptation, as it was available on all computers and operating systems used in the project at the time of implementation and is still supported by the hardware manufacturers.

OpenCL has several advantages over “standard” CPU code:

- Hardware independency
  The C-based instruction code used for OpenCL is mostly hardware independent and may run on CPUs, GPUS or FPGAs. When “new” hardware is available, it may be used without code modification, if OpenCL is supported by the driver
- On-the-fly compilation
  All required functions are compiled by the OpenCL driver at runtime in a short time (usually < 5 seconds). This also allows code adjustments to be made depending on the current calculation before it is executed, e.g. defining the number of frequencies for dimensioning the required arrays.
- OS independent
  OpenCL code can run under Windows or LINUX without changes.

3.1. Challenges using OpenCL

But there are also a number of challenges, some of which only became apparent during implementation:

- no dynamic memory support
  The size of all required arrays etc. must be known in advance and the data must be provided in one large memory block.
- no C++ object classes / no inheritance
  All required data must be transferred to special “GPU”-compatible structures that are processed with separate function calls.
- no support for complex numbers
  All functions requiring complex data had to be rewritten.
- no recursive function calls
  The hit-check and ray tracing routines had to be re-implemented without use of recursion.
- very few local memory
  The private memory / register set of a single streaming processor is very small (approx. 1 ... 4 KB), so the size of all local variables had to be minimized as far as possible in order to avoid outsourcing this data to the much slower global memory of the GPU device.
- low double precision performance
  As the performance of OpenCL on GPUs is usually at least twice as high for single as for double precision, the code was only implemented for simple accuracy to reduce the amount of data and speed up data exchange between host and GPU.
  It must be ensured that the size ratios between the evaluation distance and element “size” (edge length etc.) match when using single precision; as a rule of thumb, a ratio of approx. 1:1000 (1 m structure size for 1000 m evaluation distance) should be considered.

3.2. Main “Disadvantage” of using GPUs for raytracing

The biggest challenge while implementing the algorithm was the so-called synchronized instruction execution per computing unit (SIMD, Single Instruction Multiple Data). This means that identical code execution takes place on all streaming processors (SPs) of one compute group (CU). This behaviour has the following consequences:

- Different program paths due to decisions or branches force the SPs to wait until the end of the different code in each case.

- Looking at ray tracing, it becomes clear that no reliable prediction can be made about the number of “child” rays traced based on the “hits” of the elements (and thus the number of decisions in the code).
- A few “complex” beam trees within one CU lead to significant delays for all “simple” trees. If, for example, only 16 complex trees occur within a CU with 128 SPs, all remaining 112 SPs must “wait” until their processing is complete.
- For the most efficient utilization of the GPU, the number of decision paths in the code must therefore be as low as possible.

4. RESULTS

All results of the test cases shown were determined using a BeTSSi submarine model (length approx. 62 m) with different numbers of elements. It contains internal structures to which various boundary conditions (**BCs**, elastic shells, R/T factors, rigid) have been assigned.

All calculations were carried out for the monostatic case (sound source position = evaluation position) over the specified number of frequencies using start beams with a predefined “width” or one start beam per element. The figures showing the TES results are intended only to clarify the respective test case.

The deviations between the results of the CPUs and GPUs were within the scope of the calculation accuracy (single precision). The factors given indicate the speed advantage of the GPU compared to the CPU “reference solution”. Due to the limited space available, the results are not discussed in detail here but evaluated in the conclusions.

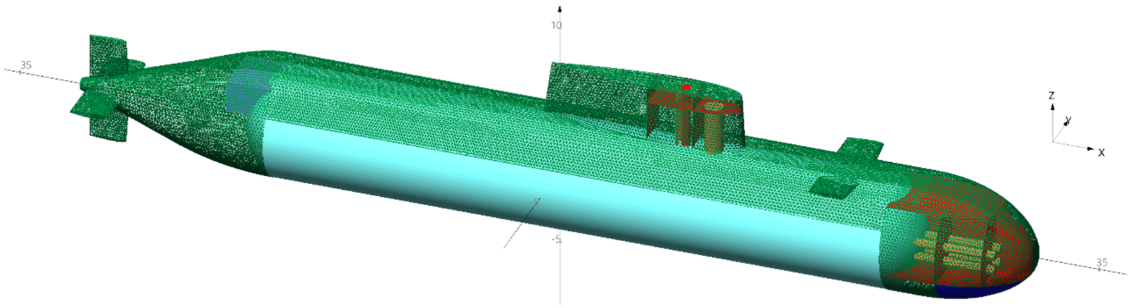
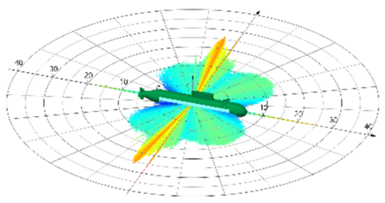


Fig. 4: BeTSSi model with inner structures (116,126 elements, only for illustration)

Test case 1: 1,8M elements, XY plane, R/T conditions

beam width:	0.2 m (max. start beam count $N_{sb,max}$: 17,160)
elements:	1,858,016
frequencies:	21
BCs:	R = 0.5, T = 0.5
evaluation:	XY-plane, 72,001 points, ±180°, 0.005° steps

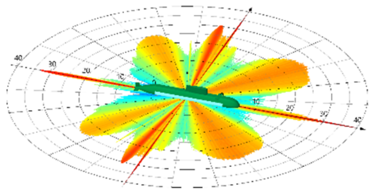


System / Hardware	Solver / mode	Cores / SPs	Threads / CUs	Solver time [s]	Factor
AMD Ryzen Threadripper Pro	BEAM (AVX2)	32	64	1,491.8	1.00
AMD EPYC 9655	"	96	192	924.0	1.61
AMD Radeon RX 7900 XTX	BEAM-OCL/1D	6,144	32	520.0	2.81
nVidia GeForce RTX 4070 SUPER	"	7,168	16	457.0	3.26
nVidia GeForce RTX 4090	"	16,384	16	235.3	6.34
nVidia GeForce RTX 4090	BEAM-OCL/2D	16,834	32/1	179.8	**8.30**

Table 4: Solution times for test case 1

Test case 2: 464K elements, XY plane, mixed boundary conditions

beam width:	one per element
elements:	464,504
frequencies:	21
BCs:	mixed, realistic (shells, R/T-factors, rigid)
evaluation:	XY-plane, 36,001 points, ±180°, 0.01° steps

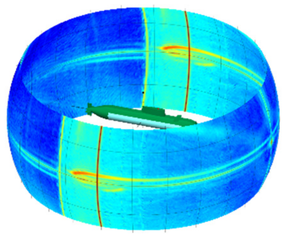


System / Hardware	Solver / mode	Cores / SPs	Threads / CUs	Solver time [s]	Factor
AMD Ryzen Threadripper Pro	BEAM (AVX2)	32	64	5,434.6	1.00
AMD EPYC 9655	"	96	192	3,142.0	1.73
AMD Radeon RX 7900 XTX	BEAM-OCL/1D	6,144	32	1,278.1	4.25
nVidia GeForce RTX 4070 SUPER	"	7,168	8	2,324.4	2.34
nVidia GeForce RTX 4090	"	16,384	8	1,504.9	3.61
"	BEAM-OCL/2D	"	64/1	563.6	**9.64**

Table 5: Solution times for test case 2

Test case 3: 1.8M elements, spherical section, R/T conditions

beam width:	0.2 m (max. start beam count $N_{sb,max}$: 17,160)
elements:	1,858,016
frequencies:	21
BCs:	R = 0.5, T = 0.5
evaluation:	spherical section, 72,821 points, ±180° / ±25°, 0.5° steps

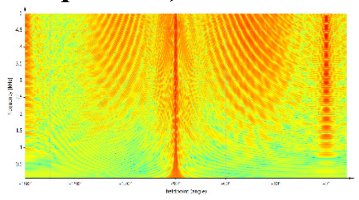


System / Hardware	Solver / mode	Cores / SPs	Threads / CUs	Solver time [s]	Factor
AMD Ryzen Threadripper Pro	BEAM (AVX2)	32	64	3,110.0	1.00
AMD EPYC 9655	"	96	192	1,656.6	1.88
AMD Radeon RX 7900 XTX	BEAM-OCL/2D	6,144	32	2,558.8	1.22
"	BEAM-OCL/2D/R4	"	"	2,310.0	1.35
nVidia GeForce RTX 4090	BEAM-OCL/2D	16,834	128/1	1,203.1	2.58
"	BEAM-OCL/2D/R4	"	"	1,094.0	**2.84**

Table 6: Solution times for test case 3

Test case 4: 464K elements, XY plane, "waterfall" (multiple frequencies)

beam width:	one per element
elements:	464,504
frequencies:	491 (vertical axis)
BCs:	mixed, realistic (shells, R/T-factors, rigid)
evaluation:	XY-plane, 3,601 points, ±180°, 0.1° steps,



System / Hardware	Solver / mode	Cores / SPs	Threads / CUs	Solver time [s]	Factor
AMD Ryzen Threadripper Pro	BEAM (AVX2)	32	64	11,429.0	1.00
AMD EPYC 9655	"	96	192	7,280.0	1.57
AMD Radeon RX 7900 XTX	BEAM-OCL/2D, 5 FB	6,144	226/1	3,912.0	2.92
nVidia GeForce RTX 4090	BEAM-OCL/2D, 10 FB	16,834	226/1	1,850.3	6.18
"	BEAM-OCL/2D, 5 FB	"	"	1,210.9	**9.44**

Table 7: Solution times for test case 4

5. CONCLUSIONS AND OUTLOOK

The implementation for the GPU leads to the following conclusions:

- The achievable acceleration of a process on GPUs basically depends on the “similarity” of the task to be solved. Performing identical tasks with different data is the optimum.
  This means for ray tracing:
  The smaller the “differences” between the beam paths, e.g. due to smaller distances between the evaluation points, the greater the speed advantage. Resorting the evaluation points according to their distance (case 3, spherical section) led to a speed advantage of ≈ 10%.
- The use of so-called two-dimensional kernel functions (recognizable in the result tables by the addition “/2D” in the Solver column), where one call is made for each start/evaluation point combination, leads to a significantly better utilization of the GPU cores.
  With the one-dimensional kernel functions tested initially, one call is made for each evaluation point, which led to a significantly poorer utilization of the GPU, especially if there are fewer evaluation points than cores.
- Overall, depending on the application, the best GPU achieved a speed advantage by a factor of 2.5 … 10 in relation to the CPU reference.
- Considering the price of the hardware, a corresponding GPU offers a better price-performance ratio, as no high-performance CPU is required in this case.

However, high-performance CPUs continue to offer advantages if the required memory space, e.g. for matrix-based methods such as BEM, exceeds the memory space available on the GPU and are generally also more flexible in use.

Surprisingly, the speed factor of the CPU with 96 cores is only ≈ 1.6 … 1.9, whereas at least 3 was expected, also due to the 5th core generation. Therefore, it is planned to explore whether the insights gained from the GPU implementation can be transferred to the CPU code version.

6. PROJECT PARTNER

This project is funded by the Bundeswehr Technical Center for Ships and Naval Weapons, Maritime Technology and Research (WTD 71), Eckernförde, Germany.

REFERENCES

[1] **R. Burgschweiger, I. Schäfer, M. Ochmann and B. Nolte**, “Results of the ray-tracing based solver BEAM for the approximate determination of acoustic backscattering from thin-walled objects”, in *Internoise 2014*, Melbourne, Australia, pp. 1 … 10, 2014.

[2] **I. Schäfer and B. Nolte,** “Berechnung der akustischen Rückstreustärke von Unter-wasserobjekten mit Hilfe der Randelementmethode und der Kirchhoffschen Hoch-frequenznäherung” in *DAGA 2008*, Dresden, Germany, 2008.

[3] **R. Burgschweiger, I. Schäfer, D. Sachau and J. Ehrlich**, “High-performance calculation of the acoustic backscattering strength based on the Kirchhoff high-frequency approximation”, in *Underwater Acoustic Conference and Exhibition (UACE 2023)*, Kalamata, Greece, pp. 303 … 310, 2023.

[4] **Khronos Group**, “OpenCL for Parallel Programming of Heterogeneous Systems”, https://www.khronos.org/opencl/

[5] **Wikipedia**, “OpenCL”, https://en.wikipedia.org/wiki/OpenCL