Simple and efficient GPU parallelization of existing $\mathscr{H}$-Matrix accelerated BEM code

Kerstin Vater[a,b], Timo Betcke[b], Boris Dilba[c]

[a]*Hamburg University of Technology, Dynamics Group, Schloßmühlendamm 30, 21073 Hamburg, Germany*
[b]*Department of Mathematics, University College London, Gower Street, London WC1E 6BT, UK*
[c]*Novicos GmbH, Kasernenstraße 12, 21073 Hamburg*

Abstract

In this paper, we demonstrate how GPU-accelerated BEM routines can be used in a simple black-box fashion to accelerate fast boundary element formulations based on Hierarchical Matrices ($\mathscr{H}$-Matrices) with ACA (Adaptive Cross Approximation). In particular, we focus on the expensive evaluation of the discrete weak form of boundary operators associated with the Laplace and the Helmholtz equation in three space dimensions. The method is based on offloading the CPU assembly of elements during the ACA assembly onto a GPU device and to use threading strategies across ACA blocks to create sufficient workload for the GPU. The proposed GPU strategy is designed such that it can be implemented in existing code with minimal changes to the surrounding application structure. This is in particular interesting for existing legacy code that is not from the ground-up designed with GPU computing in mind.

Our benchmark study gives realistic impressions of the benefits of GPU-accelerated BEM simulations by using state-of-the-art multi-threaded computations on modern high-performance CPUs as a reference, rather than drawing synthetic comparisons with single-threaded codes. Speed-up plots illustrate that performance gains up to a factor of 5.5 could be realized with GPU computing under these conditions. This refers to a boundary element model with about 4 million unknowns, whose $\mathscr{H}$-Matrix weak form associated with a real-valued (Laplace) boundary operator is set up in only 100 minutes harnessing the two GPUs instead of 9 hours when using the 20 CPU cores at disposal only. The benchmark study is followed by a particularly demanding real-life application, where we compute the scattered high-frequency sound field of a submarine to demonstrate the increase in overall application performance from moving to a GPU-based ACA assembly.

Keywords: BEM, GPU computing, Hierarchical Matrices, sonar cross section

1. Introduction

Many problems in engineering science referring to an equilibrium state in a homogeneous medium can be modeled by the Boundary Element Method (BEM). The fundamental idea is to express the solution only in terms of values on the boundary of the calculation domain. This feature makes the BEM especially suitable for unbounded domains, as they are frequently encountered in acoustics, electrostatics and fluid dynamics.

In the course of a boundary element simulation, a system of linear equations is set up to find the unknown part of the boundary data. For this purpose, it is necessary to evaluate the discrete weak formulations of the boundary operators involved. Their explicit computation is usually the most expensive part of a boundary element simulation, since classically $O\left(N^2\right)$ integrals have to be evaluated numerically for a BEM problem with N elements. Hierarchical Matrices ($\mathscr{H}$-Matrices) based on the Adaptive Cross Approximation (ACA) approach can reduce this complexity to $O\left(N \log N\right)$. Together with the relative simple implementation of ACA it has therefore become the method of choice for many large-scale industrial applications.

With the rise of Graphics Processing Units (GPUs) to be used for scientific computations, entirely new possibilities have opened up to further accelerate boundary element simulations. Graphics cards have been originally designed for image rendering purposes, where tremendous amounts of light-weight, independent tasks

---

Email addresses: `kerstin.vater@tuhh.de` (Kerstin Vater), `t.betcke@ucl.ac.uk` (Timo Betcke), `dilba@novicos.de` (Boris Dilba)

are processed in parallel. It turns out that numerical integration routines in the BEM operate in a very similar way, in the sense that they are also massively invoked and execute in a Single Instruction Multiple Data (SIMD) fashion. This circumstance suggests that GPU computing may benefit the assembly procedure of the discretized weak form of boundary operators in the BEM, and therefore further speed up the expensive setup of the equation system.

The acceleration of boundary element simulations has been a subject of research for decades. Until about 2010, activities in this area were mainly focused on the development of fast approximation algorithms. Popular approaches such as the Fast Multipole Method (FMM) [1, 2, 3] or $\mathcal{H}$-Matrices [4, 5] have proven capable of significantly reducing the computational effort associated with boundary element calculations.

One of the first attempts to speed up the BEM using graphics hardware has been made in 2009 by Takahashi and Hamada [6] when they presented a GPU-accelerated BEM formulation based on the GPU programming platform CUDA to solve the Helmholtz equation in three dimensions. Two years later, Labaki et al. [7, 8] published a review on three distinct GPU implementations addressing two-dimensional potential and electrostatic problems.

In 2011, Yokota et al. [9] employed the BEM to investigate biomolecular electrostatic interactions governed by a Poisson equation. For this purpose, they designed a solver using GPU hardware acceleration on top of an FMM code. Computational experiments with billion-scale problems demonstrated good parallel efficiency on up to 512 GPUs. Shortly after, the authors implemented an auto-tuning mechanism [10], which enabled reasonable scaling of their FMM code on up to 4000 GPUs in the context of particle-based turbulence simulations.

For the GPU acceleration of $\mathcal{H}$-Matrix compression Christophersen proposed in 2012 an OpenCL-based algorithm that used interpolation techniques for the far-field and not ACA. Recently, Börm and Christophersen [11] suggested a hybrid adaptive algorithm based on a technique called Green's cross approximation to achieve very good GPU performance. However, this technique makes a redesign of the $\mathcal{H}$-Matrix code necessary.

The main objective of this paper is to reveal how existing $\mathcal{H}$-Matrix BEM codes can be upgraded with GPU computing to accelerate the dense and $\mathcal{H}$-Matrix weak-form assembly of all four commonly encountered boundary operators associated with the three-dimensional Laplace and Helmholtz equations. These are the single- (SLP) and double-layer potential (DLP)

operators, as well as the adjoint double-layer potential (ADLP) and hypersingular (HYPS) boundary operators. We will show that it is actually reasonable to apply GPU computing within the flexible framework of an existing BEM library framework running on a desktop computer with minimum code changes, and give realistic impressions of what can be expected from this approach.

For this purpose, the GPU programming interface CUDA C/C++ will be employed, which is included in the CUDA Toolkit [12] version 8.0. The implementation is based on an experimental branch of the Galerkin boundary element library Bempp [13] version 3.1 (www.bempp.com), which provides flexible solutions for various BEM applications in a black-box manner. Therefore, the possibility to choose boundary elements of variable polynomial order will be preserved on the GPU. The system under consideration is a workstation equipped with two Intel Xeon processors E5-2670 v2 [14] with ten physical cores each, operating at 2.5 GHz processor base frequency, as well as two NVIDIA GeForce GTX TITAN Black boards [15] with a full Kepler GK110 GPU implementation each, and about 200 GB of system memory.

The remainder of this paper is structured as follows. Section 2 briefly reviews the mathematical framework of $\mathcal{H}$-Matrix-based Galerkin boundary element formulations. Sections 3 and 4 discuss how the relevant parts of the underlying C++ code can be adapted such that the dense and $\mathcal{H}$-Matrix-based assembly routines run on GPUs. Benchmark results in Section 5 illustrate the performance of the proposed algorithms for various Laplace and Helmholtz boundary operators, where both the complete and the $\mathcal{H}$-Matrix weak-form construction are investigated. We conclude the benchmarks with a realistic industry example to compute the scattered sound field of a submarine. In Section 7, we present a summary and an outlook on ongoing work.

2. Galerkin-based boundary integral operators and their discretization

In this work, we consider Galerkin discretizations of the single-layer potential (SLP), double-layer potential (DLP), adjoint double-layer potential (ADLP), and hypersingular (HYPS) boundary operators for three-dimensional Laplace and Helmholtz problems. The cor-

responding discrete matrices are defined as follows.

$$[S_h]_{i,j} = \int_{\Gamma_j} \int_{\Gamma_i} \psi_i(x) \, g(x,y) \, \phi_j(y) \, ds_y ds_x, \qquad (1)$$

$$[K_h]_{i,j} = \int_{\Gamma_j} \int_{\Gamma_i} \psi_i(x) \, \frac{\partial g(x,y)}{\partial n_y} \phi_j(y) \, ds_y ds_x, \qquad (2)$$

$$\left[K'_h\right]_{i,j} = \int_{\Gamma_j} \int_{\Gamma_i} \psi_i(x) \, \frac{\partial g(x,y)}{\partial n_x} \phi_j(y) \, ds_y ds_x. \qquad (3)$$

$$[D_h]_{i,j} = \int_{\Gamma_j} \int_{\Gamma_i} g(x,y) \Big[\mathrm{curl}_\Gamma \psi_i(x) \cdot \mathrm{curl}_\Gamma \phi_j(y)$$

$$-k^2 \psi_i(x) \, n_x \cdot \phi_j(y) \, n_y \Big] ds_y ds_x. \qquad (4)$$

In the above equations, n denotes the unit normal vector at a specified point on the boundary Γ of the solution domain, and ϕ_i and ψ_j represent real basis functions of the domain and test space, respectively. We note that for the hypersingular operator the functions ϕ_j and ψ_i must be at least continuous and piecewise linear. The above definition for D can then be obtained from the normal derivative of the double-layer potential, followed by integration by parts.

The Green's function of the Helmholtz equation $(-\Delta u - k^2 u = 0)$ in 3D is defined as

$$g(x,y) = \frac{\exp(ik|x-y|)}{4\pi|x-y|}. \qquad (5)$$

The Laplace equation in 3D $(-\Delta u = 0)$ has the same Green's function with the wavenumber k set to zero.

2.1. Hierarchical Matrices

In the following, we briefly review ACA-based $\mathscr{H}$-Matrix assembly [16] of boundary operators. The basic idea is to approximate the fully-populated matrices (1) to (4) by using only a few of the matrix entries. For this purpose, a tree-based geometric partitioning is introduced. Leafs of the tree correspond to matrix blocks, which are either admissible with respect to a distance and block diameter dependent admissibility condition, or inadmissible. Admissible blocks can be compressed. Inadmissible blocks are stored in dense mode. An example partitioning is shown in Figure 1. Large blocks are admissible while most of the very small blocks are inadmissible.
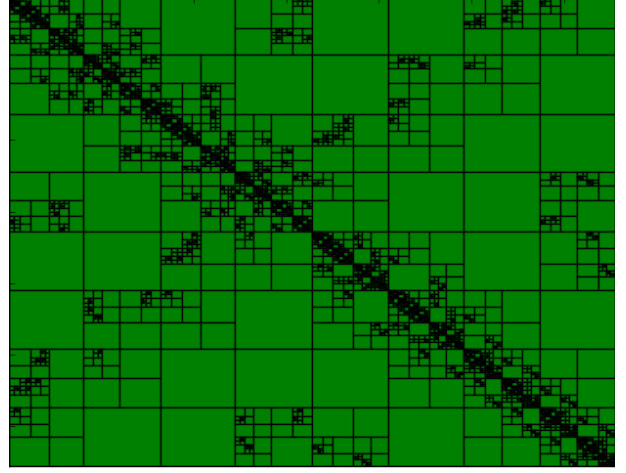

Figure 1: Hierarchical partition of a matrix depending on a geometric admissibility condition.

Several different approaches exist to treat the admissible blocks. The probably most popular one is known as the Adaptive Cross Approximation (ACA) procedure. An introduction and analysis of the ACA algorithm is given in Bebendorf [16, Section 3.4]. In the following, we briefly recap the principle of ACA as it will be important to understanding the GPU algorithm later on.

Consider an admissible leaf block $A \in \mathbb{R}^{m \times n}$. Starting from $R_0 := A$, the recursion formula

$$R_{k+1} := R_k - \left[(R_k)_{i_k, j_k}\right]^{-1} (R_k)_{1:m, j_k} (R_k)_{i_k, 1:n} \qquad (6)$$

yields a low-rank representation of the block A. The index ranges $(R_k)_{i, 1:n}$ and $(R_k)_{1:m, j}$ denote the i-th row and the j-th column of R_k, respectively. Essentially, this means that we have to find a non-zero pivot (i_k, j_k) in R_k, which is used as a scaling factor for an outer product of the i_k-th row and the j_k-th column. The result is then subtracted from R_k to get R_{k+1}, and so forth.

The column index j_k is chosen by finding the maximum element of the current row i_k according to

$$\left|(R_{k-1})_{i_k, j_k}\right| = \max_{j=1,\dots,n} \left|(R_{k-1})_{i_k, j}\right|. \qquad (7)$$

The selection of the row index i_k, however, turns out to be more complicated. It is described in detail by Bebendorf [16, Section 3.4.3].

On closer inspection of the recursion (6) it becomes apparent that the k-th step only incorporates matrix entries in the j_k-th column and the i_k-th row to compute a better approximation R_{k+1}. Thus, it is not necessary to determine the full matrix R_k in each step, except from a few of the original entries of A. Taking advantage of this property, the ACA algorithm takes the following form.

procedure ACA (admissible leaf block $A \in \mathbb{R}^{m \times n}$)

    Initialization $k \leftarrow 1$, $Z \leftarrow \emptyset$

    repeat

        Determine row index i_k

        Compute selected row $\tilde{v}_k \leftarrow a_{i_k,1:n}$

        ▷ Subtract scaled preceding results

        for $l = 1, \ldots, k-1$ **do**

            $\tilde{v}_k \leftarrow \tilde{v}_k - (u_l)_{i_k} \, v_l$

        end for

        Add i_k to Z

        if $\tilde{v}_k$ does not vanish **then**

            Find column index $j_k \leftarrow \max_{j=1,\ldots,n} \left| (\tilde{v}_k)_j \right|$

            $v_k \leftarrow (\tilde{v}_k)_{j_k}^{-1} \, \tilde{v}_k$

            Compute selected column $u_k \leftarrow a_{1:m,j_k}$

            ▷ Subtract scaled preceding results

            for $l = 1, \ldots, k-1$ **do**

                $u_k \leftarrow u_k - (v_l)_{j_k} \, u_l$

            end for

            $k \leftarrow k + 1$

        end if

    until stopping criterion is met

        or Z contains all m rows

end procedure

Here, u_k denotes the j_k-th column of R_{k-1}, i.e. $(R_{k-1})_{1:m,j_k}$, and $\tilde{v}_k$ is the i_k-th row of R_{k-1}, i.e. $\left[(R_{k-1})_{i_k,1:n} \right]^T$. The matrix $S_k := \sum_{l=1}^{k} u_l v_l^T$ eventually represents an approximation of $A = S_k + R_k$.

The rank of S_k is bounded by the number of updates k. The rank also determines the accuracy of the approximation. Therefore, a maximum number of iterations $k_{\max}$ can be defined adaptively as a stop criterion, such that a predefined level of accuracy is ensured. However, the algorithm usually converges already after a few steps.

Note that for practical application the prototype ACA algorithm as outlined above is extended by several features, such as heuristic sub-block detection and zero-block identification.

3. Dense GPU parallelization

In the following section, we turn to the major concern of this paper, the development of simple and effective GPU-accelerated weak-form assembly routines. First, we will give a very brief introduction into GPU computing with CUDA. We then describe the host program managing the deployment of accelerator hardware, as well as the actual GPU integration routines. We begin with the boundary operators as introduced in Section 2,

where the discrete weak formulation is represented by a fully-populated matrix. The resulting code parts will serve as the basis for the following $\mathscr{H}$-Matrix weak-form assembly.

3.1. GPU programming with CUDA

The term "CUDA" originally stood for "Compute Unified Device Architecture", and is essentially a parallel computing platform with a dedicated application programming interface (API) developed by NVIDIA since the late 1990s. The programming model enables scientists and engineers to access graphics processors for GPGPU purposes in a straightforward way. The CUDA API is built on top of the high-level programming language C/C++, which is commonly used for scientific software development and High Performance Computing purposes.

The general concept of GPU programming with CUDA is that of a *host* program executing on the CPU that manages computational tasks performed on *devices*. Therefore, the terms CPU/host and GPU/device are used interchangeably in the following.

In order to get work done by the GPU, the host invokes a special function called *kernel*. A CUDA kernel is characterized by the fact, that it is called from the host program and executes on the device. Each parallel thread on the GPU then launches an instance of the kernel, where work is spread across the requested compute resources via process indices. Detailed CUDA programming guides are freely available from the NVIDIA website [17, 18], and will therefore not be explicated here.

3.1.1. Hardware architecture of GPUs

For code development we have used TITAN Black graphics cards which are based on the NVIDIA Kepler GK110 microarchitecture [19], which hosts 15 Streaming Multiprocessor Units (SMX). Each SMX features 192 single precision CUDA cores, as well as 64 double precision units. This "precision gap" is the reason, why calculations in single precision typically run much faster on GPUs than double precision computations. The amount of fast accessible on-chip memory is restricted to 48 KB assigned to the L1 cache, 8 KB read-only data cache, and 1536 KB L2 cache memory. Note that the cache of a GPU multiprocessor is generally very small compared to modern CPU configurations. For comparison, one core of the Intel Xeon processors E5-2670 v2 has 25 MB cache at its disposal. This fact makes an efficient memory management on the GPU essential for good performance.

3.1.2. Thread parallelism in CUDA

In CUDA, parallel tasks are mapped to grids of blocks with blocks of threads. The described multiprocessors schedule tasks in *warps* of 32 threads each, which execute in a SIMD fashion. Hence, branching in individual threads inside a warp (branch divergence) leads to slow down as CUDA runs through both branches in an if/then/else statement and discards results as necessary. Thus, GPU computing is made for identical, parallelizable tasks.

3.2. Host code

Each element of the global BEM matrix consists of sums of integrals over test and trial elements. The numerical evaluation of these raw weak-form integrals is clearly the most expensive part of the matrix assembly procedure. Therefore, we source out this subroutine to the GPU, while the subsequent summation of element contributions to the respective degrees of freedom in the matrix is left to the CPU.

3.2.1. Basic procedure

The host code of the GPU-accelerated dense-matrix weak-form assembly can be outlined as follows. First, the total amount of work associated with all pairs of test and trial elements is distributed evenly over the available devices. This approach assumes that the GPUs which are attached to the system are all of the same type, which is usually the case for modern high-performance workstations and computer clusters.

The participating devices are covered by a parallel loop, where each GPU runs through the following steps. First, the respective device is initialized. This procedure includes the transfer of the raw grid data with vertex coordinates and element definitions to the global device memory space. Since in BEM we deal with surface grids which are typically not very large in size we always transfer the complete grid information to a device. Geometry information such as normal vectors and Jacobians are computed on the device and stored as 1D structures according to the principle of coalesced access to global device memory [17, Section 5.3].

Moreover, the relevant basis functions (and if necessary also their spatial derivatives) are evaluated at the quadrature points defined with respect to the unit triangle. These values reside in the global device memory space, as well.

Numerical quadrature weights, however, are stored in constant memory. These are only a few values, which are accessed by all GPU threads at the same time. Hence, this type of memory suggests itself. One practical drawback of constant memory is that it needs to be defined as a static array, the size of which must be known at compile time. Therefore, we impose a limit of six double-precision floating-point numbers in the first place, which corresponds to a quadrature order of 4 on plane triangular boundary elements using a symmetric Gauss triangle quadrature rule.

After the initialization process is completed, each device subdivides its work package further into chunks of element pairs. In this way, the evaluation of unassembled matrix entries on the GPU can be overlapped with the host-side global assembly of elementary contributions to keep the available hardware busy and decrease the total execution time.

The unassembled GPU results related to a chunk are stored in *zero-copy* host memory. This type of memory is advantageous in this case, because only two result buffers of the size corresponding to one chunk need to be allocated in the beginning of the process. Furthermore, integral values are written back only once at the end of the CUDA kernel in a coalesced fashion. Thus, zero-copy memory is supposed to be the fastest approach.

The chunks of element pairs are now processed successively. In doing so, the numerical integration and the global assembly procedure are considered as two distinct tasks, where the former is basically handled by the GPU and the latter is assigned to the CPU. These two tasks are part of a parallel TBB [20] task group and can execute simultaneously. This means that one chunk passes through the global assembly process on the CPU, while the next bunch of element pairs is already integrated on the device. This approach helps to keep all available hardware as busy as possible.

Nevertheless, it turns out that either task necessarily limits the overall performance. Obviously, results can be assembled only after the computation of the corresponding values has been completed. This is ensured through a barrier between the integration step and the assembly step associated with the same chunk.

The described procedure is summarized in Figure 2, as well as in the pseudo code below:

```
procedure DENSE ASSEMBLY
    Allocate result matrix A
    Distribute integrals over devices
    parfor all devices do
        Initialize device
        Set up parallel task group
        for all chunks of integrals b do simult.
            ▷ Integrate bᵢ on device and wait
            parfor all integrals rⱼ in bᵢ do
                rⱼ ← GPU results
            end parfor
            ▷ Assemble bᵢ₋₁ into matrix A on host
            parfor all integrals rⱼ in bᵢ₋₁ do
                if rⱼ is singular then
                    A ← r̃ⱼ (CPU results)
                else if integral is regular then
                    A ← rⱼ (GPU results)
                end if
            end parfor
        end for
        Wait for last chunk to finish assembly
    end parfor
    return matrix A
end procedure
```

3.2.2. Global matrix assembly

During the global assembly procedure, the raw data are finally merged, i.e. the contributions from individual boundary elements are added to the global degrees of freedom. Since multiple parallel processes may attempt to add their integral values to the same matrix entry at the same time, the access to the corresponding memory locations needs to be controlled via *mutexes*. With the aim to minimize the inactive times when threads are waiting for their turn, every single matrix entry is protected by its own mutex variable. Alternative strategies employ only one mutex for the whole matrix, or column- and row-wise mutexes, respectively. However, our fine-grained approach has proven most effective. Note that this is not an issue with piecewise constant basis functions. In this case, matrix entries are assigned to exactly one pair of elements, thus parallel threads do not influence each other.

3.2.3. Handling of different types of integrals

To obtain sufficiently accurate evaluations of the discrete BEM kernels between test and trial elements suitable quadrature routines need to be chosen. If the test and trial element are disjoint a standard Gauss triangle
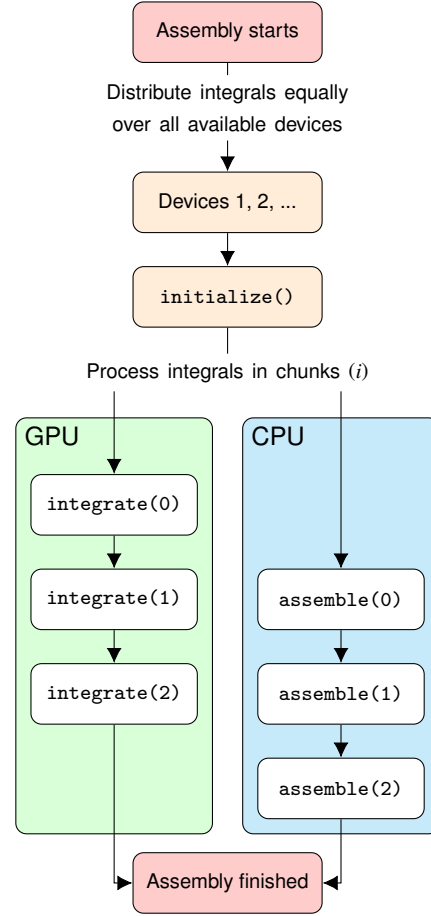

Figure 2: Dense weak-form assembly procedure

quadrature rule is sufficient. For neighboring triangles (sharing a vertex or an edge) or in the case that the test and trial element are identical, suitable transformations need to be performed to remove the singularities. In our implementation we use a fully numerical scheme proposed by Erichsen and Sauter [21].

Only results related to regular integrals of disjoint elements are taken from the GPU computation. Otherwise, element pairs assigned to threads of a warp may choose different execution paths according to suitable integration routines, which leads to branch divergence on the device. This violates the SIMD paradigm of the GPU hardware architecture, and therefore harms performance. A possible remedy would be to presort the element pairs with respect to their integration scheme, and then call one CUDA kernel for each batch separately. However, the necessary preparations are not only cumbersome, but also fragment the work package resulting in multiple less efficient kernel calls. For this reason, we forgo singular integration on the device in this work.

Instead, values related to singular integrals are simply overridden with cached results that have been evaluated on the CPU in advance.

3.2.4. Data precision

Due to the often significantly higher single precision performance on GPUs our implementation allows to switch the data type on the device between single and double precision. One argument is the restricted accuracy of the Gauss quadrature, which usually does not exceed single-precision accuracy.

3.2.5. Treatment of complex numbers

The last comment concerns the handling of complex-valued (Helmholtz) boundary integral operators. In this case, the real and imaginary part of complex numbers are consequently separated on the device. Although complex data types are available in CUDA, for instance as a part of the Thrust Parallel Algorithms Library [22], Hawick and Playne [23] have already shown that calculating and storing real and imaginary parts separately should be preferred over a compound data type for the sake of performance.

3.3. GPU implementation of the numerical integration routine

In the implementation of the GPU integration routine, the given task is split up into two subsequent steps, the evaluation of kernel function values and the summation procedure. These are processed within a single CUDA kernel call, where one thread is assigned to one pair of elements.

Since all element-related input data such as normal vectors, global quadrature points, and Jacobian determinants have been precalculated in advance, these information are simply loaded to thread-local memory. The BEM kernel function is then evaluated for all pairs of quadrature points, where the intermediate results are stored locally. Afterwards, the actual numerical integration procedure is triggered. The kernel function values are multiplied by quadrature weights, and basis functions values are incorporated for the different combinations of local degrees of freedom. Finally, the results are written back to global device memory in a coalesced fashion.

---

procedure INTEGRAL EVALUATION (element pair (m, n))
    Reconstruct test and trial element indices from thread index
    Load input data into thread-local device memory
    for all pairs of Gaussian points (x_m, y_n) **do**
        Evaluate kernel function $g(x_m, y_n)$
    end for
    for all combinations of local dofs (d_m, e_n) **do**
        Evaluate integral $r(d_m, e_n)$
        Store results $b \leftarrow r(d_m, e_n)$
    end for
end procedure

---

4. GPU acceleration of the H-Matrix assembly

Implementing ACA-based $\mathscr{H}$-Matrix compression efficiently on the GPU is not straight forward. The iterative ACA scheme with per-block adaptive error control leads to bad load balancing and branch divergence issues inside a warp. A possible solution is to perform only the actual integral evaluation on the GPU, and keep the complex flow control on the CPU. However, this means sending only a single row or column of a block to the device at a time, leading to excessive kernel calls or underutilization. In [11] an approach is suggested in which a thread traverses the $\mathscr{H}$-Matrix tree and adds blocks to a task list. Once a task list is sufficiently large, another thread sends it to the GPU device for processing. While this approach is effective, it is implementationally difficult with significant bookkeeping and thread management and requires reorganization of the existing $\mathscr{H}$-Matrix code, which we aim to avoid here.

In our proposed GPU-accelerated $\mathscr{H}$-Matrix assembly each block is handled by one CPU thread. These parallel threads send integration jobs directly to the device without switching the context such that all data stays local to one CPU thread and any bookkeeping is omitted. Traditionally, the kernel calls on the GPU would have been serialized by the CUDA device and then executed one after another, leading again to underutilization problems as each thread only sends a row or column. The solution is to make use of NVIDIA's Hyper-Q technology. Hyper-Q was introduced with the Kepler generation of NVIDIA GPUs and increases the number of hardware work queues from one to thirty-two. This allows multiple CPU threads or processes to send smaller amounts of data to the device whereas the independent work queues ensure that device utilization remains high.

7

Moreover, inter-thread communication is avoided. This approach is comparatively simple to implement and it fits well with existing CPU-based $\mathscr{H}$-Matrix implementations which usually use threading to split up blocks between different CPU cores. The implementation of the row and column computation can simply build on the existing dense GPU integrator from Section 3.

4.1. Basic procedure

The GPU implementation of the $\mathscr{H}$-Matrix weakform assembly comprises the following steps. First, the tree-based block partitioning of the $\mathscr{H}$-Matrix is generated on the CPU. This is a purely geometry-based computation and is very fast. Then, the participating GPUs are initialized. The grid data is copied to the device memory of the GPU, where element-related data is precalculated and cached in the global device memory. Afterwards, a thread-parallel loop over all $\mathscr{H}$-Matrix blocks is performed. Small non-admissible blocks involving singular integrals are treated in dense mode on the CPU. The remaining admissible blocks making up the largest part are fed to the ACA algorithm. For smaller blocks the ACA is completely performed on the CPU as the memory transfer overhead would be too large. The rows and columns of larger blocks are sent to the GPU during the course of the ACA assembly. The following pseudo code demonstrates this algorithm (see also Figure 3.

---

procedure H-Matrix Assembly
    Declare H-Matrix A
    Generate block cluster tree
    parfor all leaf blocks b **do**
        if b is not admissible **then**
            Evaluate block in dense mode on CPU
        else if b is admissible **then**
            Approximate block using ACA
            if b is large enough **then**
                Compute rows/columns of b on GPU
            else if b is small **then**
                Compute rows/columns of b on CPU
            end if
        end if
        H-Matrix $A \leftarrow$ block b
    end parfor
    return H-Matrix A
end procedure

The advantage is that the code is very similar to an existing pure CPU thread-parallel ACA assembly routine.
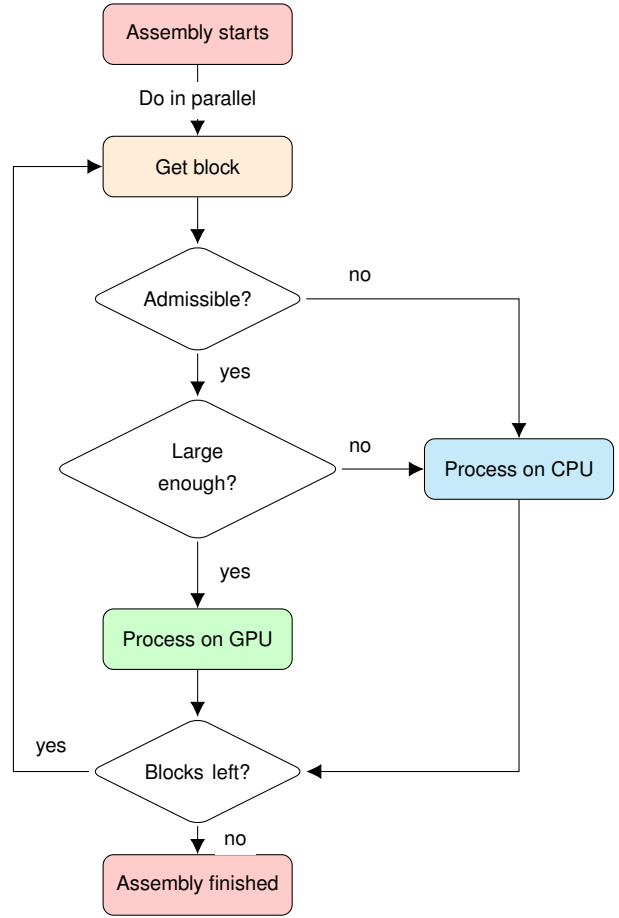

Figure 3: $\mathscr{H}$-Matrix weak-form assembly procedure

The difficult work of scheduling the rows and columns on the GPU is automatically performed through the hardware worker queues. Software management of the workload is not necessary.

If multiple GPUs are used we employ a simple round-robin strategy that passes the work packages around the participating GPUs. The local $\mathscr{H}$-Matrix assembler works in a very similar way to the dense assembler. First, the work package consisting of test-trial element pairs is split up into smaller chunks if their number exceeds a user-defined maximum. After that, memory is allocated both on the host and on the device to store the unassembled results. In contrast to the dense-matrix weak-form assembly, we use simple *pageable* host memory and global device memory in this case. Therefore, results need to be copied to the host system explicitly, as we can not rely on the implicit way in conjunction with pinned host memory and zerocopy strategies. The reason for these different choices of

host memory is that during the $\mathscr{H}$-Matrix construction memory is allocated and deallocated frequently, every time a row or column of a block needs to be computed. In the dense-matrix case this is done only once at the beginning and at the end of the whole procedure. Since allocation and deallocation of pinned host memory is quite expensive, it is way more efficient to stick with ordinary pageable host memory in this case.

Next, the test and trial element indices are copied to the global device memory, and a parallel task group is created to perform the numerical integration on the device simultaneously with the assembly of previously calculated values. A loop over all chunks finally constructs the requested part of a matrix block.

Note that the employed numerical integration routines are exactly the same as in the dense-matrix assembly case. The corresponding CUDA kernel has been described in detail in Section 3.3.

4.2. A remark on compiler parameters

It is important to add the `-default-stream per-thread` compiler flag to the `nvcc` command. This essentially enables the Hyper-Q technology and ensures that each host thread accessing a GPU creates its own default stream [24], and thus multiple host-controlled calculations can be performed on one device concurrently.

5. Realistic performance benchmarks

5.1. Hard- and software configuration

The benchmark tests are based on an experimental GPU enabled version of Bempp 3.1 (`www.bempp.com`). The code is run on a desktop workstation equipped with two Intel Xeon processors E5-2670 v2 [14] operating at 2.5 GHz processor base frequency, as well as two NVIDIA GeForce GTX TITAN Black GPUs [15] with a Kepler GK110 GPU each. We use the CUDA Toolkit [12] version 8.0. The size of the system memory is about 200 GB. We note that the aim of this section is to perform realistic benchmark tests. Hence, we are not interested in comparing the GPU setup against a single CPU thread computation, but the overall performance of the GPU setup compared to a modern dual Xeon workstation that is typical for many BEM application scenarios.

5.2. Experimental setup and procedure

We determine the performance gains over the multi-threaded CPU-optimized version of the code in terms of

a speed-up factor

$$S = \frac{t_{\mathrm{CPU}}}{t_{\mathrm{GPU}}}, \qquad (8)$$

where t_{CPU} is the execution time of the CPU and t_{GPU} is the execution time for the GPU enabled algorithms presented in the previous sections.

As benchmark problem we consider a unit sphere, which is discretized with various numbers of plane triangular boundary elements. Both the trial and the test space are restricted to globally continuous, piecewise linear basis functions. Note that basically all types of boundary elements provided by the Bempp library are supported on GPUs as a result of this work. However, other types of boundary elements do not introduce any significantly new aspects here. The number of unknowns N arise from convenient mesh sizes h, where the smallest value of h is chosen to exploit the available resources (i.e. the host memory) as much as possible. Our workstation features about 200 GB system memory. We then compute the discretized weak formulations of the three-dimensional Laplace and Helmholtz boundary operators as introduced in Section 2, where both dense and $\mathscr{H}$-Matrices are employed. Each configuration is run 5 times to give averaged performance results.

5.3. Dense-matrix weak-form assembly results

Figure 4 summarizes the computational results for the dense-matrix assembly of boundary operators. Initially, the speed-up rises steeply as the problem size grows, since any fixed overhead arising from the involvement of GPUs is spread over a larger number of unknowns. Then, it gradually levels off, and even shows a slight decrease in most cases. This can be ascribed to a reduced GPU clock frequency due to an increased mean GPU temperature under load. As heat is known to adversely affect performance when power consumption is held constant [25], NVIDIAs GPU Boost 2.0 technology [26] applies dynamic overclocking with a temperature target of 80 °C. This feature can not be disabled for GeForce GTX TITAN graphics cards operated under Debian Linux to enforce a constant clock frequency. The speed-up is likely to go up again when the GPUs eventually operate at their base frequency. However, due to memory limitations this can not be shown here.

As expected, GPU computations in single precision perform significantly faster than in double precision. This is particularly pronounced for complex-valued (Helmholtz) boundary operators. Our graphics boards provide three times as many single-precision floating-point units per multiprocessor as double-precision
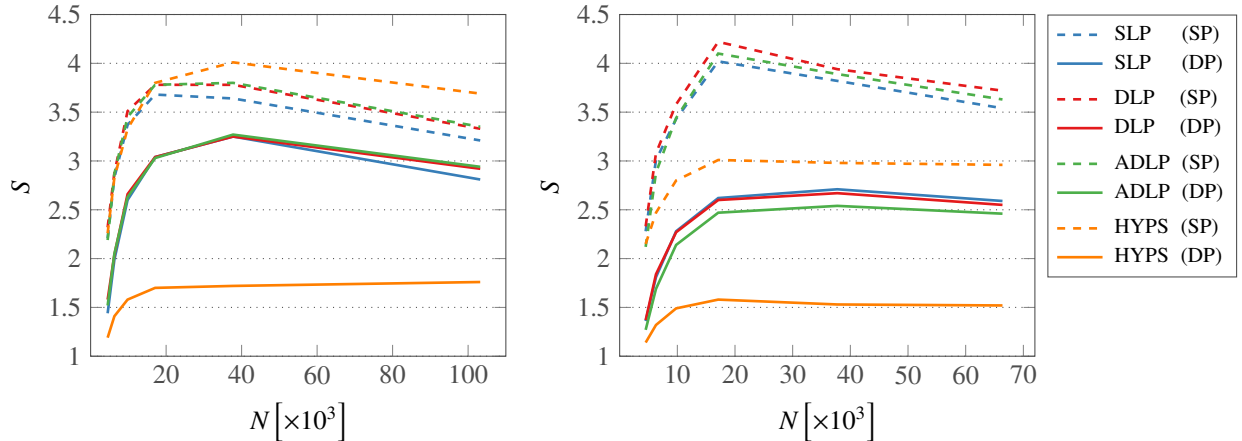
Figure 4: Speed-up values S for the dense weak-form assembly of Laplace (left) and Helmholtz (right) boundary operators depending on the number of unknowns N in single precision (dashed lines) and double precision (solid lines)

cores, which partly translates into the performance results. Issues of memory access and data transfer do not allow for an equivalent speed-up factor, though.

In comparison with the SLP, DLP and ADLP boundary operators, the HYPS boundary operator shows considerably less satisfactory results. Unlike the other three operators, the integral evaluation associated with the hypersingular operator requires the computation of an inner vector product at the lowest level of the nested quadrature loops. Our kernel codes are not well adapted to handle the corresponding memory access pattern, since we have not optimized this further. The reason is that we can exploit the weak-form shown in (4) to write the HYPS operator in the form

$$D = \sum_{j=1}^{3} Q_j^T \hat{S} Q_j - k^2 \sum_{j=1}^{3} P_j^T \hat{S} P_j, \qquad (9)$$

where $\hat{S}$ is the representation of the single-layer potential operator in a space of discontinuous, element-wise linear basis functions, the matrices Q_j are sparse matrices that represent the j-th curl-component of the trial functions, and the P_j are sparse matrices that correspond to the product of the basis functions with the j-th component of the normal vector. While this formulation is not interesting for dense assembly, it provides benefits for the more application relevant $\mathscr{H}$-Matrix assembly since $\mathscr{H}$-Matrix assembly of the single-layer operator on a discontinuous space is in pure CPU experiments already about twice as fast as the assembly of the matrix D. However, the price is that the memory requirements grow by about a factor six due to the increased overall matrix size. More details to this approach and numerical experiments can be found in [27].

5.4. H-Matrix weak-form assembly results

In the first part of this section we have focused on the assembly of the discrete weak form represented by a fully populated matrix. We now turn over to the fast approximation of boundary operators based on $\mathscr{H}$-Matrices. As in the previous section, both the Laplace and the Helmholtz equation are considered. The results are depicted in Figure 5.

The kink in the line graphs substantiate the assumptions from the dense-assembly study that the slight decline in the speed-ups in Figure 4 is a local phenomenon, rather than a global trend for the GPU-accelerated dense-matrix computation. The problem sizes, for which the mentioned significant changes in the underlying conditions occur (GPU clock rate), happen to coincide with the memory limit for dense-matrix weak-form representations. Here, the local depression is overcome, as much larger problems can be handled by means of the $\mathscr{H}$-Matrix approach.

Although the deviating behavior of the hypersingular operator has already been addressed in the course of the first part of this section, it is very striking here that the speed-ups related to single precision GPU computations tend to be lower than the factors obtained in the double precision case. This effect becomes stronger for larger problem sizes. An explanation can be found with a view to the number of iterations in the ACA algorithm. It turns out that, unlike the other operators, the HYPS operator is strongly affected by the precision level of the numerical integration on the GPU in the sense, that the approximation procedure for the admissible leaf blocks converges more slowly. Therefore, the total amount of work increases as more original matrix
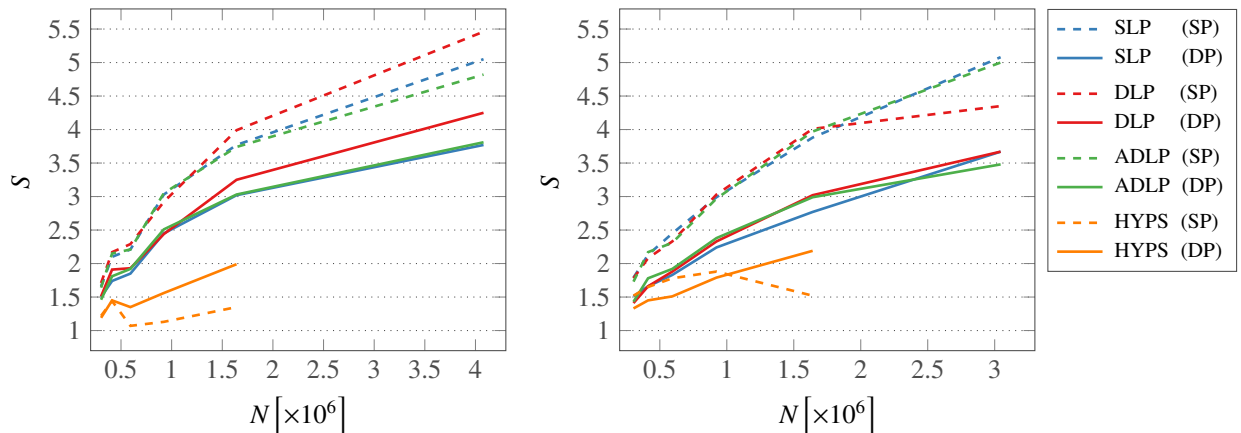
10

Figure 5: Speed-up values S for the $\mathscr{H}$-Matrix weak-form assembly of Laplace (left) and Helmholtz (right) boundary operators depending on the number of unknowns N in single precision (dashed lines) and double precision (solid lines)

entries need to be computed. This can not be compensated by a faster numerical integration process in single precision, thus the $\mathscr{H}$-Matrix construction slows down as a whole compared to double precision computation. As a consequence, low precision numerical integration should be generally avoided in conjunction with hypersingular boundary operators where the weak form is based on $\mathscr{H}$-Matrices.

Note that the maximum speed-up that could be observed in Figure 5 is limited by the number of unknowns N, and therefore the size of the system memory in this case. However, the shape of the curves suggest that way higher factors can be realized for large-scale problems, provided that enough memory is available to store the $\mathscr{H}$-Matrix weak form.

6. High-frequency scattering of a submarine hull

In order to demonstrate the benefits of our newly developed GPU-accelerated $\mathscr{H}$-Matrix BEM code to complex real-world problems we consider an exterior Helmholtz problem. In particular, we investigate the high-frequency scattered field from a plane wave impinging on a sound-hard submarine hull. To this end, a special OSRC-preconditioned Burton-Miller formulation [28, 29, 30] is applied to compute the *bistatic target strength* of the submarine. This quantity indicates how detectable an object is with sonar in some distance, where the angular position of the transmitter and the receiver of the signal may differ. The ability to simulate the target strength characteristics is a prerequisite to develop construction features that give the submarine a desired low or high reflection coefficient already at an early stage in the design process.

The incident plane wave can be modeled as

$$u^{inc}(x) = u_0 \exp(ik\langle d, x\rangle), \qquad (10)$$

where u_0 is the amplitude, d is the direction vector, k denotes the wave number, and x specifies a point in space.

The OSRC Burton-Miller formulation to compute the missing Dirichlet boundary data of the total field $\Phi = u^{sct} + u^{inc}$ reads

$$\left(\frac{1}{2}I - K - \tilde{V}_{NtD}D\right)\Phi = u^{inc} - \tilde{V}_{NtD}\frac{\partial u^{inc}}{\partial n_x}. \qquad (11)$$

Here, $\tilde{V}$ is an on-surface approximation to the Neumann-To-Dirichlet operator. Further details on this OSRC preconditioning are given, for instance, by Antoine and Darbas [28].

Using the double-layer potential operator K, the scattered acoustic field simply yields

$$u^{sct} = K\Phi. \qquad (12)$$

We define a measure for the deviation of the GPU results from the CPU values based on n evaluation points in the far field as

$$\Delta_{sct} = \frac{1}{n}\sum_{i=1}^{n}\frac{\|u_{GPU}| - |u_{CPU}\|}{|u_{CPU}|}. \qquad (13)$$

Finally, the bistatic target strength is defined as

$$TS_{dB} = 20\log_{10}\left(R\left|\frac{u^{sct}}{u_0}\right|\right), \qquad (14)$$

with distance R from the geometrical center of the investigated object.

11

In our example, we set the frequency of the incoming wave to $f = 1000\,\text{Hz}$ and apply $c_{\text{water}} = 1500\,\text{m s}^{-1}$, such that the wave number $k \approx 4.2$. Assuming that 6 to 10 boundary elements per wavelength $\lambda = \frac{c}{f}$ is enough to ensure good accuracy, we obtain a mesh with $M = 230\,000$ linear plane triangles and $N = 115\,000$ unknowns. This happens to be the largest problem that can be treated on our workstation in terms of system memory. The incident plane wave of amplitude $u_0 = 1\,\text{N m}^{-2}$ strikes the hull laterally at a $\theta^{\text{inc}} = 10°$ angle from the submarine's longitudinal axis, thus the direction vector is defined as $d = \left[\cos\left(\theta^{\text{inc}}\right), \sin\left(\theta^{\text{inc}}\right), 0 \right]^T$. Further, we position 3600 evaluation points at a distance of $r = 20\,\text{km}$ from the geometrical center of the submarine model.

Figure 7: Bistatic target strength TS depending on the angular position θ^{sct} at a distance $r = 20\,\text{km}$ from the submarine's center

Figure 7 shows the target strength of the submarine.

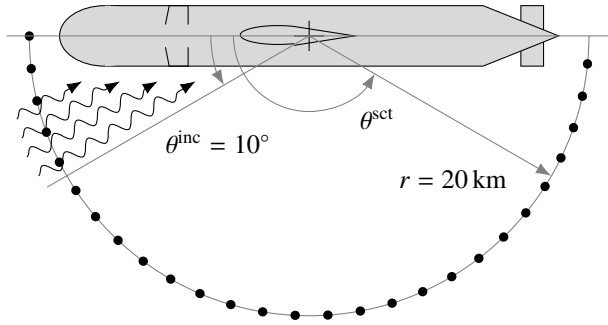Figure 8: Dirichlet boundary data of the total field $|\Phi|$

Figure 6: Experimental setup of the scattering submarine hull

The threshold for rows or columns of $\mathcal{H}$-Matrix blocks to be treated by graphics processors is set to $10\,000$ participating element pairs. All GPU computations are performed in single precision. The assembly of the hypersingular operator D was implemented as demonstrated in (9) through sparse transformations of the single-layer boundary operator on a space of discontinuous linear functions. Direct assembly of the HYPS operator did not yield sufficient results in single-precision.
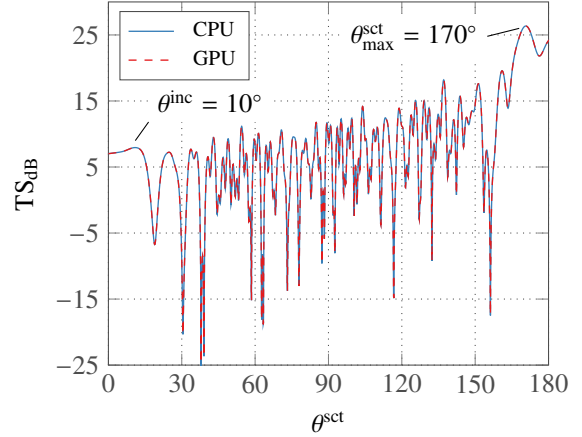
Figure 8 visualizes the absolute values of the Dirichlet boundary data that has been solved for in equation (11).

Using the GPU-accelerated assembly routines, the setup of the equation system took only $1206\,\text{s}$, compared to $2275\,\text{s}$ with the multi-threaded CPU-optimized code. This translates into a speed-up factor of 1.9 for the discrete weak-form assembly of the required boundary operators, while the deviation between GPU and CPU results as defined by equation (13) yields a negligible value of $\Delta_{\text{sct}} = 0.0081\,\%$. It is interesting to note how the assembly speed-up translates into a speed-up for the total application. The problem was solved in 18 iterations due to the effectiveness of the OSRC preconditioner in this application. The assembly of all sparse operators and the solution of the iterative system were performed purely on the CPU. The total time (exclud-

ing reading the grid) including assembly, iterative solver and evaluation of the bistatic target strength with the GPU enabled code was 2473 s while the time for the pure CPU code was 3397 s, a total speed-up of a factor of 1.4. Hence, the more the assembly dominates the total solution time the more will an application benefit from the simple GPU strategy described in this paper.

7. Conclusions

The aim of this paper was to develop a simple strategy to benefit from accelerated GPU computations for existing BEM codes without requiring a substantial rewrite. Indeed, only the core integration routines need to be replaced by corresponding GPU routines while the rest of the $\mathscr{H}$-Matrix assembly and surrounding code remains mostly unchanged. This allows an easy upgrade path for existing BEM codes to take advantage of GPU compute capabilities.

The performance advantage is considerable. A setup of gaming level dual Kepler generation boards outperformed a pure 20-core Xeon CPU workstation by a factor of 1.9. This means that with modest investments in hardware and software development existing BEM codes can achieve significant speed-ups for a wide range of applications.

We note that while other authors have suggested GPU-accelerated $\mathscr{H}$-Matrix compression (see e.g. [11]), the approach taken here achieves good performance improvements without major rewrites by simply replacing the core assembly routines. The key difference is the availability of NVIDIA's Hyper-Q technology which simplifies code design for GPUs significantly. Moreover, we have provided a realistic application study that demonstrates how the assembly speed-up translates over from synthetic benchmarks to realistic application problems that additionally involve sparse operators and preconditioners, which are assembled on the CPU.

Further acceleration can be achieved by translating the complete tool chain including sparse operators and iterative solvers to the GPU. However, this requires substantial investments in redesigning the software infrastructure while at the same time classical CPU workstations are also increasing their speed considerably with modern high-end dual Xeon workstations providing over 50 physical CPU cores. We believe that the proposed algorithm in this paper provides a good balance between modest investments in hardware and software development while still achieving very good assembly speed-ups that translate into a noticeable overall application performance improvement.

Acknowledgement

References

[1] L. Greengard, V. Rokhlin, A Fast Algorithm for Particle Simulations, Journal of Computational Physics 73 (1987) 325–348.

[2] Y. Liu, Fast Multipole Boundary Element Method: Theory and Applications in Engineering, Cambridge University Press, 2009.

[3] S. Keuchel, K. Vater, O. v. Estorff, hp Fast Multipole Boundary Element Method for 3D Acoustics, International Journal for Numerical Methods in Engineering 110 (2017) 842–861.

[4] S. Börm, L. Grasedyck, W. Hackbusch, Introduction to Hierarchical Matrices with Applications, Engineering Analysis with Boundary Elements 27 (2003) 405–422.

[5] W. Hackbusch, A Sparse Matrix Arithmetic Based on\ cal h-Matrices. Part I: Introduction to {\ Cal H}-Matrices, Computing 62 (1999) 89–108.

[6] T. Takahashi, T. Hamada, GPU-Accelerated Boundary Element Method for Helmholtz' Equation in Three Dimensions, International Journal for Numerical Methods in Engineering 80 (2009) 1295–1321.

[7] J. Labaki, L. S. Ferreira, E. Mesquita, Constant Boundary Elements on Graphics Hardware: A GPU-CPU Complementary Implementation, Journal of the Brazilian Society of Mechanical Sciences and Engineering 33 (2011) 475–482.

[8] J. Labaki, E. Mesquita, S. L. Ferreira, The BEM on General Purpose Graphics Processing Units (GPGPU): A Study on Three Distinct Implementations, in: Proceedings of the 11th International Conference on Boundary Element Techniques XI. United Kingdom: EC Ltd Press, pp. 316–323.

[9] R. Yokota, J. P. Bardhan, M. G. Knepley, L. A. Barba, T. Hamada, Biomolecular Electrostatics Using a Fast Multipole BEM on up to 512 GPUs and a Billion Unknowns, Computer Physics Communications 182 (2011) 1272–1283.

[10] R. Yokota, L. Barba, T. Narumi, K. Yasuoka, Scaling Fast Multipole Methods up to 4000 GPUs, in: Proceedings of the ATIP/A* CRC Workshop on Accelerator Technologies for High-Performance Computing: Does Asia Lead the Way?, A* STAR Computational Resource Centre, p. 9.

[11] S. Börm, S. Christophersen, Approximation of BEM Using GPGPUs, CoRR abs/1510.07244 (2017).

[12] NVIDIA, CUDA Toolkit, https://developer.nvidia.com/cuda-toolkit, 2017. Accessed: 2017-11-02.

[13] W. Śmigaj, T. Betcke, S. Arridge, J. Phillips, M. Schweiger, Solving Boundary Integral Problems with BEM++, ACM Transactions on Mathematical Software (TOMS) 41 (2015) 6.

[14] Intel, Intel Xeon Processor E5-2670 v2 Specifications, https://ark.intel.com/products/75275/Intel-Xeon-Processor-E5-2670-v2-25M-Cache-2_50-GHz, 2017. Accessed: 2017-11-02.

[15] NVIDIA, NVIDIA GeForce GTX TITAN Black, http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan-black, 2017. Accessed: 2017-11-02.

[16] M. Bebendorf, Hierarchical Matrices, Springer, 2008.

[17] NVIDIA, CUDA C Programming Guide, https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf, 2017. Accessed: 2017-11-02.

[18] NVIDIA, CUDA C Best Practices Guide, https://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf, 2017. Accessed: 2017-11-02.

[19] NVIDIA, NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110, `https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf`, 2012. Accessed: 2017-11-02.

[20] Intel, Threading Building Blocks (TBB), `https://www.threadingbuildingblocks.org/`, 2017. Accessed: 2017-11-02.

[21] S. Erichsen, S. A. Sauter, Efficient automatic quadrature in 3-d Galerkin BEM, Computer Methods in Applied Mechanics and Engineering 157 (1998) 215–224. Papers presented at the Seventh Conference on Numerical Methods and Computational Mechanics in Science and Engineering.

[22] J. Hoberock, N. Bell, Thrust Parallel Algorithms Library, `https://thrust.github.io/`, 2017. Accessed: 2017-11-02.

[23] K. A. Hawick, D. P. Playne, Numerical Simulation of the Complex Ginzburg-Landau Equation on GPUs with CUDA, in: Proc. IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN), pp. 39–45.

[24] M. Harris, GPU Pro Tip: CUDA 7 Streams Simplify Concurrency, `https://devblogs.nvidia.com/parallelforall/gpu-pro-tip-cuda-7-streams-simplify-concurrency/`, 2015. Accessed: 2017-11-02.

[25] D. Price, Fire and Ice: How Temperature Affects GPU Performance, `http://on-demand.gputechconf.com/gtc/2014/presentations/S4484-how-temperature-affects-gpu-performance.pdf`, 2014. GPU Technology Conference.

[26] NVIDIA, GPU Boost 2.0 Technology, `http://www.nvidia.de/object/nvidia-gpu-boost-2-de.html`, 2017. Accessed: 2017-11-02.

[27] T. Betcke, M. W. Scroggs, W. Śmigaj, Product algebras for Galerkin discretizations of boundary integral operators and their applications, in preparation (2017).

[28] X. Antoine, M. Darbas, Generalized Combined Field Integral Equations for the Iterative Solution of the Three-dimensional Helmholtz Equation, ESAIM: Mathematical Modelling and Numerical Analysis 41 (2007) 147–167.

[29] M. Darbas, E. Darrigrand, Y. Lafranche, Combining analytic preconditioner and Fast Multipole Method for the 3-D Helmholtz equation, Journal of Computational Physics 236 (2013) 289–316.

[30] T. Betcke, E. van 't Wout, P. Gélat, Computationally Efficient Boundary Element Methods for High-Frequency Helmholtz Problems in Unbounded Domains, Birkhäuser, Cham, 2017, pp. 215–243.