GPU-Accelerated Method of Moments by Example: Monostatic Scattering

Evan Lezar* and David B. Davidson†
Computational Electromagnetics Group
Department of Electrical and Electronic Engineering
Stellenbosch University
Stellenbosch, South Africa
*mail@evanlezar.com †davidson@sun.ac.za

Abstract

In this paper we combine and extend two of our previous works to provide a more complete solution for the GPU-acceleration of the Method of Moments using CUDA by NVIDIA. To this end, the formulations of the original 1982 Rao–Wilton–Glisson paper are revisited and the scattering analysis of a square PEC plate is considered as a simple example. One of the primary contributions of the paper is to serve as a guide for the implementation of other GPU accelerated computational electromagnetic routines and as such provides a background on general purpose GPU computation as well as insight into the finer details of the implementation. The results computed compare well with reference values, and from a performance point of view the GPU implementation is found to be significantly faster. The fastest measured speedup for one of the phases of the Method of Moments computations was more than $140\times$. This translates into a speedup of about $45\times$ when considering the entire Method of Moments solution process for the problem considered.

Index Terms

Boundary element methods, Electromagnetic scattering, Parallel programming, Parallel architectures, Linear algebra, NVIDIA CUDA, General purpose GPU computing

I. Introduction

One of the techniques that is widely used in the modelling and analysis of electromagnetic scattering and radiation phenomena is the method of moments (MOM) or boundary element method. One of the reasons for this is that the MOM incorporates the correct behaviour for the radiation condition and does not require that free-space be discretised [1]. Although acceleration methods such as the multi-level fast multi-pole method (MLFMM) exist, these are not applicable to all practical problems, and at times a standard MOM implementation is required.

The use of GPUs to perform computational tasks not necessarily related to the graphics processing for which they were originally designed has been around in some form or another since the late 1970s [2]. The end of 2006 saw great improvements both in terms of usability and performance. This was mainly due to the introduction of the G80 architecture and CUDA by NVIDIA [3], [4] and CTM (later the Stream SDK) by ATI/AMD [5]. Since then devices by both vendors have improved greatly with better double precision support and peak single precision performance exceeding 2 teraFLOPS (TFLOPS). In this paper we leverage the computational power of CUDA-capable GPUs to accelerate the method of moment analysis of a simple electromagnetic scattering problem.

The application of GPU acceleration using CUDA to the MOM problem is presented in [6], although in that case acoustic problems are considered and the matrix assembly formed part of an iterative solution scheme for the linear system that needs to be solved. In this work the matrix is first assembled after which direct solution methods are employed this allows for the matrix factorisation to be reused – which is especially useful in scattering problems such as the one considered here. The results presented here rely on native double precision computation, whereas in [6] only single precision and double-single precision [7] results are presented. In earlier papers by Chen, Xu, and Ding [8], and Peng and Nie [9] electromagnetic scattering problems are considered, although they use OpenGL Shading Language [10] and Brook [11] respectively, and not NVIDIA CUDA as is the case here and are as such restricted to single precision implementations.

Two works that do employ CUDA and matrix factorisation methods to solve the MOM problem in computational electromagnetics are [12] and [13], although both once again only consider single precision implementations. Further, in [12] pulse basis functions are used to model a two-dimensional scattering problem and not a generalized scattering problem based on the Rao–Wilton–Glisson basis functions [14] as considered here. In [13] roof-top basis functions are used along with an integration scheme that can only be used in a rectangular mesh with equal cells [15] to solve problems involving planar structures such as micro-strip patch antennas. As is the case in this work, the MAGMA library [16] is also used to perform the LU decomposition. The authors of [13] also present an investigation into the CUDA acceleration of the finite difference time domain (FDTD) method in [17], where the regular grid and similar operations performed on each Yee cell [1] lend themselves well to GPU acceleration.

This work sees the combination of two prior studies by the authors, namely [18] and [19], where two of the phases in the MOM solution process have been considered independently. Although the method and results of [14] are also recreated in [18], this paper provides a much more in-depth discussion of the development process. In [19] one of the phases is considered purely from a linear algebra point of view and as such this work is the first direct application of the presented results to the field of computational electromagnetics. Although that work provides a panel-based approach do deal with the memory size limitations of current GPUs, this is not considered here as the nuances of the implementation are outside the scope of this paper. Instead the MAGMA library – which forms part of the solution process in [19] – is used directly.

What further sets this paper apart from previous work, is that it intends to primarily serve as a guide in the GPU acceleration of computational electromagnetic codes. To this end, the techniques outlined in one of the seminal papers in the MOM by Rao, Wilton, and Glisson [14] are used as a core component of the implementation discussed. In order to make the discussion more tangible, a simple scattering problem is considered and drives the development process. The problem also allows for the verification of any code developed.

Section II serves to introduce the reader to the problem considered as well as providing relevant background information on the method of moments and GPU computing in general. This is then built upon in the subsequent sections. In Sections III and IV the GPU implementation of two of the phases of the method of moments are discussed in some detail. The Discussion starts with the solution of the linear system, as the availability of libraries such as MAGMA [20] and CUBLAS [21] allow us to ease into GPU computing. When considering the matrix assembly in Section IV the GPU related implementation details given are more in depth. In Section V a number of results are presented that not only verify the accuracy of the implementations discussed but also provides an analysis of the performance improvements attained by using GPU acceleration.

II. Background

A. Monostatic scattering

In order to develop the discussion regarding the acceleration of the method of moments using GPUs, let us consider a simple example – monostatic scattering off a square PEC plate as depicted in Figure 1. Such a plate, measuring one wavelength on each side and located in the xy-plane, with an incident field propagating in the direction of the negative z-axis is used in [14] and in this paper.

One quantity of interest in scattering problems such as these is the monostatic radar cross-section (RCS). This involves determining the ratio of scattered power density to the incident power density at a given frequency and gives an indication of how visible an object is to radar from a specific angle [1]. Usually such a calculation is performed over a range of incident angles. The RCS of the square plate is used for verification of the implementations presented here, with the results presented in Section V.

B. The Method of Moments

Scattering problems such as this can be solved numerically using the electric field integral equation formulation of the method of moments and a simplified block diagram of the MOM solution process is given in Figure 2. The block diagram also illustrates that some parts of the process are independent of the angle of incidence (for a given frequency) thus giving an indication of where data can be reused to improve the computational efficiency of the solution.

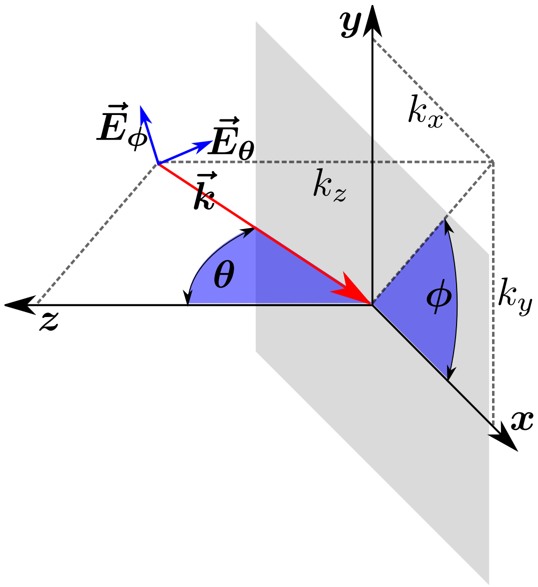

Fig. 1. A diagram showing an incident plane wave with arbitrary linear polarisation and direction of incidence ($\vec{k}$). The definitions of the angles (blue arcs) used in the calculation of the radar cross section of an object located at the origin and the square PEC plate (grey) considered in this paper are also shown.

Consider for example the calculation of the impedance matrix $[\mathbf{Z}]$. This matrix is independent of incident angle, and thus if the RCS is being calculated it is only necessary to assemble it once. Furthermore, if the LU decomposition of the matrix is performed outside the loop over the incident angles this factorisation can be used each time the linear system is solved, greatly reducing the time required to obtain a solution.

When considering the GPU-acceleration of the method of moments, each of the steps offers its own challenges and potentials for improving performance. It should also be noted that not all steps contribute equally to the total time required to obtain a solution and in general the matrix assembly and the solution of the linear system of equations make up the bulk of the computational requirement (as is highlighted in Section V) and as such, this paper focuses on their acceleration.

For the purpose of performance analysis the matrix assembly, excitation vector calculation, and linear system solution phases are considered. The initialisation and post-processing phases are not included in the timing analysis. The initialisation phase

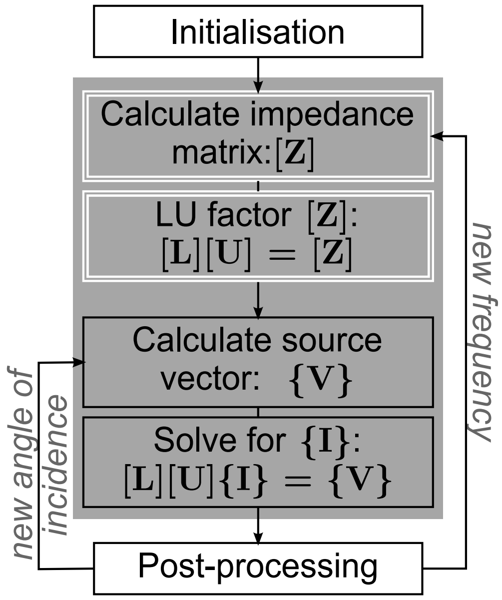

Fig. 2. A block diagram showing the steps involved in obtaining the method of moments solution for a typical scattering problem. The phases considered here for timing purposes are surrounded by a grey box and the phases for which GPU acceleration is applied also highlighted by double white outlines.

may include the meshing any objects in the computation (such as the PEC plate) and although this can be a costly operation, this only needs to be performed once after which the mesh can be stored and reloaded at a comparatively low cost. In contrast to the initialisation phase, the post-processing phase may differ greatly depending on the information required. In addition the time requirement is affected by things such as the level of detail required in the visualisation of calculated property, such as the surface current on the plate. For the post-processing phase it is also possible to store the computed vector $\{\mathbf{I}\}$ and calculate any desired properties at a later stage for a given mesh.

For the phases in the MOM process that are not considered in depth here, the reader is referred to texts such as [1] for more background.

1) Matrix assembly: As should already be evident from the block diagram in Figure 2, the method of moments can be loosely summarised as the construction of the complex valued linear system

$$[\mathbf{Z}]\{\mathbf{I}\} = \{\mathbf{V}\}, \tag{1}$$

and its subsequent solution to find the unknown current coefficients $\{\mathbf{I}\}$. In this section, the assembly of the impedance matrix $[\mathbf{Z}]$ is discussed with the solution of the linear system discussed in Section II-B2.

As an introduction to the matrix assembly process, the basis functions introduced in [14] are reviewed. To this end, consider the diagram in Figure 3 that depicts two adjacent triangles in a triangulation (mesh) of a surface and the n^{th} edge that is shared by them. Choose one of the triangles as the positive triangle and denote it as T_n^+ and the other as T_n^- (the negative triangle).

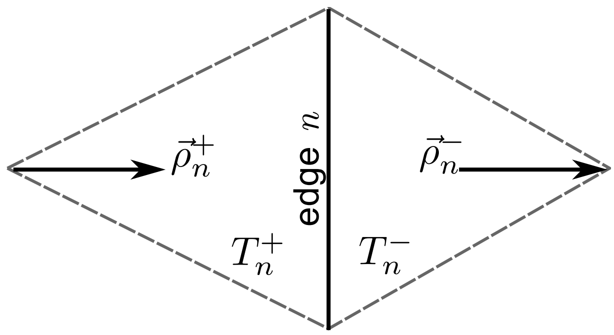

Fig. 3. A diagram of a free edge in a surface mesh as well as the designation of the positive and negative triangles. Also shown are the vectors used in the definition of the basis function associated with the edge.

The basis function associated with the n^{th} edge, $\vec{f}_n(\vec{r})$, at any point $\vec{r}$ in space is then given by [14]

$$\vec{f}_n(\vec{r}) = \begin{cases} \frac{l_n}{2A_n^+}\vec{\rho}_n^{\,+}(\vec{r}), & \vec{r} \text{ in } T_n^+ \\ \frac{l_n}{2A_n^-}\vec{\rho}_n^{\,-}(\vec{r}), & \vec{r} \text{ in } T_n^- \\ 0, & \text{otherwise,} \end{cases} \tag{2}$$

with l_n the length of the n^{th} edge, and A_n^+ and A_n^- the areas of the positive and negative triangles associated with the edge respectively. The vector $\vec{r}$ is the position where the basis function is to be evaluated and the $\vec{\rho}_n^{\,\pm}(\vec{r})$ vectors are calculated as the difference between $\vec{r}$ and the nodes opposite edge n (with orientations as indicated in Figure 3). This basis function then represents a unit current density flowing across the edge and is only non-zero in the triangles that share the edge.

With the basis functions defined, the vector current density on the surface of the mesh can be discretised and approximated by the weighted sum of the basis functions of all the triangles on the surface as follows [14]

$$\vec{J}(\vec{r}) = \sum_{n=1}^{N} I_n \vec{f}_n(\vec{r}), \tag{3}$$

where N is the number of degrees of freedom (DOFs) - equal to the number of non-boundary edges in the surface triangulation of the geometry. Solving for the unknown coefficients I_n (the elements of $\{\mathbf{I}\}$ in (1)) is the goal of the MOM solution phases highlighted in grey in Figure 2.

Using these basis function definitions the entries, Z_{mn}, of the impedance matrix $[\mathbf{Z}]$ can be calculated. In the classic RWG formulation, this entry is approximated as the combination of the magnetic vector potentials as well as the electric scalar potentials at the centres of the triangles associated with the m^{th} edge as a result of the current density associated with the n^{th} edge [14], expressed mathematically as

$$Z_{mn} = l_m \left[\frac{j\omega}{2} \left(\vec{A}^{+}_{mn} \cdot \vec{\rho}^{+}_{m} + \vec{A}^{-}_{mn} \cdot \vec{\rho}^{-}_{m} \right) - \left(\Phi^{+}_{mn} - \Phi^{-}_{mn} \right) \right]. \tag{4}$$

As such, it is required to integrate over both triangles associated with edge n. The integrands have the form

$$\vec{A}^{\pm}_{mn} = \frac{\mu}{4\pi} \int_{T_n^{\pm}} \vec{f}_n(\vec{r}) \frac{e^{-jk|\vec{r}^{c\pm}_m - \vec{r}|}}{|\vec{r}^{c\pm}_m - \vec{r}|} dT_n^{\pm}, \tag{5}$$

and

$$\Phi^{\pm}_{mn} = \mp \frac{1}{4\pi j\omega\epsilon} \int_{T_n^{\pm}} \frac{l_n}{A_n^{\pm}} \frac{e^{-jk|\vec{r}^{c\pm}_m - \vec{r}|}}{|\vec{r}^{c\pm}_m - \vec{r}|} dT_n^{\pm}, \tag{6}$$

for the magnetic vector potentials and electric scalar potentials respectively. The vectors $\vec{r}^{c\pm}_m$ are the position of the centre of the positive and negative triangles of edge m and the integrals are a result of a two-point approximation applied to the integration over these triangles [14].

Since the influence of all the edges on each other must be considered, the computational and memory demands of calculating the impedance matrix $[\mathbf{Z}]$ is $\mathcal{O}(N^2)$.

2) Solution of the linear system: As already mentioned, the method of moments reduces to solving the complex valued linear system given in (1) for the unknown current coefficients $\{\mathbf{I}\}$. As indicated in Figure 2 this step is dependent on both the input frequency as well as the angle of incidence, although the latter only affects the excitation vector $\{\mathbf{V}\}$. One of the ways with which to solve such a linear system is to make use of the LU decomposition of $[\mathbf{Z}]$ [22]. In the LU decomposition the matrix $[\mathbf{Z}]$ is decomposed into lower and upper triangular matrices ($[\mathbf{L}]$ and $[\mathbf{U}]$ respectively) as follows

$$[\mathbf{P}][\mathbf{Z}] = [\mathbf{L}][\mathbf{U}], \tag{7}$$

with $[\mathbf{P}]$ a matrix that applies a number of row interchanges to $[\mathbf{Z}]$ in order to keep the process numerically stable [22]. The matrices $[\mathbf{L}]$ and $[\mathbf{U}]$ can then be used to solve the linear system. In the case of an RCS calculation, they can be reused for each incident angle as the matrix $[\mathbf{Z}]$ (and thus its factors) does not change.

The factors $[\mathbf{L}]$ and $[\mathbf{U}]$ are used as follows to solve the desired linear system. Firstly construct the vector $\{\mathbf{b}\}$ as

$$\{\mathbf{b}\} = [\mathbf{L}]^{-1}[\mathbf{P}]\{\mathbf{V}\}, \tag{8}$$

and then solve the linear system

$$[\mathbf{U}]\{\mathbf{I}\} = \{\mathbf{b}\}. \tag{9}$$

Although the computational cost of obtaining $[\mathbf{L}]$ and $[\mathbf{U}]$ through LU decomposition is $\mathcal{O}(N^3)$, the triangular nature of these matrices result in the above operations being performed at a cost of $\mathcal{O}(N^2)$. This compares favourably to full Gaussian elimination, especially if the operations need to be performed repeatedly [22]. In this paper, only the LU decomposition is considered for GPU acceleration by using the routines provided by the MAGMA library. The steps in (8) and (9) are however included for timing purposes.

C. General purpose GPU computing

As already mentioned, the use of GPUs for general computational tasks has seen much interest of late and can be attributed to a number of reasons. The rapid increase in the peak performance of these devices is one of them [23], but perhaps more important in their adoption has been the introduction of programming environments such as the StreamSDK by ATI [5] and CUDA by NVIDIA [4] as well as the platform-independent OpenCL [24] which have greatly improved the ease with which these powerful devices can be programmed. The importance of ease of use is clearly illustrated by CUDA's strength in the market over competing offerings. For the purpose of this paper CUDA is used for the required GPU implementation.

A CUDA capable device, such as the GeForce GTX 280 (built on the GT200 architecture [25]) used to obtain the results presented in this work, can be seen as an external accelerator to the host. The device has its own memory hierarchy and cannot, in general, access the host memory directly [23]. Thus, before any computation can take place on the device, device memory must be allocated and the required input data transferred from the host memory to the device memory. After the desired computation has been performed on the device and the results are available in device memory, these must then be

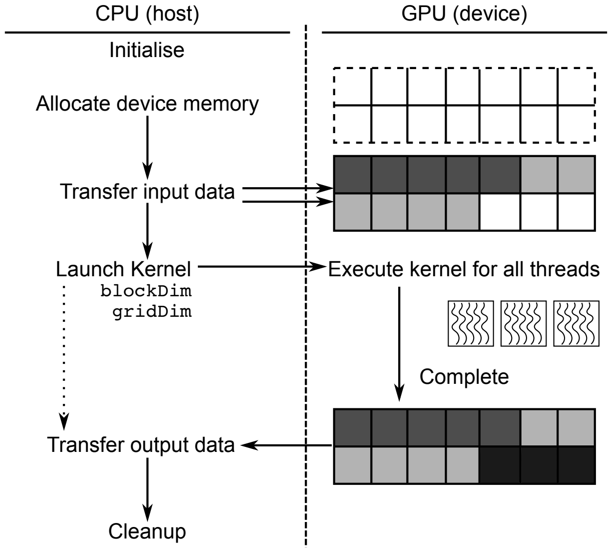

Fig. 4. A diagram showing the program flow of a typical CUDA application. `blockDim` and `gridDim` represent the dimensions of each block and the grid and are used for kernel execution.

transferred back to the host before they can be used further on the host. This data flow is represented schematically in Figure 4. Since the cost in transferring data to and from the device must also be taken into account when considering the performance of any GPU implementation, algorithms that perform a large number of operations per byte transferred are better suited to GPU acceleration.

One of the primary concerns when designing CUDA was that the resultant architecture be scalable. This is especially important in a competitive market such as consumer graphics as there are a number of segments that need to be addressed separately. A scalable architecture means that a single design can be used as the basis for products ranging from low-performance entry-level devices to high-end devices such as the GTX 280 considered here.

Although the smallest computational unit on such devices is a CUDA core or streaming processor (SP) – a single 32bit floating point and integer processor – it makes more sense to start discussions at the level of the symmetric multiprocessor (SM). Each SM is made up of eight SPs and in the GT200 architecture also includes two special functions units (SFUs) and a double precision (64bit) floating point unit (DP) [25]. There is also a small shared memory area that can be used as a user-managed cache. Although each SM functions independently, multiple SMs are grouped into a thread processing cluster (TPC) with instruction scheduling and register hardware being shared between them [25]. In the case of the GT200 architecture each TPC is made up of three SMs, whereas with the G80 architecture there are only two SMs per TPC. A TPC is then the smallest independent unit from a hardware production point of view, with a low-end device containing only one TPC (16 to 24 CUDA cores). In a high-end device such as the GTX 280, ten TPCs are used resulting in a device with 240 CUDA cores.

In order to utilise the large number of cores at its disposal, CUDA provides the ability to manage thousands of lightweight GPU threads in hardware which can be scheduled and with zero overhead [26]. In order to organise these threads from both a programming and execution point of view, they are grouped into blocks that can be one, two, or three-dimensional and the blocks can then further be grouped into a one or two-dimensional grid [23]. Each of the threads can then be uniquely identified based on its position in the global grid (using its position in a block and that blocks position in the grid) and this information can be used to decide which part of a large data set a particular thread is responsible for processing [26]. This is illustrated further with reference to the problem considered here in Section IV-C. Each of these threads executes the same device code called a kernel, which is the code responsible for performing a desired computation on the device, and is typically unique to the problem that is being considered.

The arrangement of threads into blocks in a grid further provides CUDA implementations with a scalability on hardware of differing capabilities. This is achieved by assigning each of the blocks in the grid to an SM for execution where the required instructions are executed by the SPs (or SFUs or DP) for each of the threads in the block in a SIMD fashion [26]. If a group of threads are stalled – when waiting for a high latency memory transfer for example – another group of threads is scheduled (at zero overhead) to continue their execution. This is termed *latency hiding* and is one of the reasons why it is important to ensure that data sets are big enough to ensure that a large number of threads are used. Since a block of threads executes on a single SM, the shared memory of the SM can be used as a cache that allows for communication between the threads in a block. If the number of blocks exceeds the number that can be handled by the SMs on the device at once, additional blocks are assigned once blocks complete their execution of the kernel.

It should be noted that although the discussion here focuses on CUDA, many of the concepts – especially from the software point of view – have equivalents in other languages such as OpenCL and hardware implementations such as those of AMD/ATI. In terms of the software, the idea of a CUDA thread, block, and grid can be replaced by a work item, work group, and global work size respectively [27]. This direct equivalence is to be expected since a CUDA device is also capable of executing OpenCL code. On the hardware side, a direct comparison is more difficult as the NVIDIA and AMD/ATI use vastly different architectures. The concepts of CUDA SPs, SMs, and TPCs can however be loosely mapped to a processing elements, stream cores, and compute units, respectively [28]. As a result of these similarities, the implementations discussed here could also be successfully ported to OpenCL with similar performance improvements.

An in-depth discussion of AMD/ATI hardware and OpenCL is outside the scope of this article, and as such the reader is referred to sources such as [28] and [24]. In addition to providing an introduction to OpenCL, [26] also provides a lot of information on CUDA – both in terms of the hardware and software architecture and the development process. Another excellent introduction to CUDA development concepts is [29] while the CUDA C Programming Guide provides a decent background on CUDA in general [23].

III. Implementation: Solving the linear system

As discussed in Section II-B2, the linear system that results as part of the method of moments can be solved using an LU decomposition of the impedance matrix $[\mathbf{Z}]$. One of the widely accepted libraries that perform this kind of decomposition (and others) is the LAPACK [30] library. With regards to GPU computing, there are two main offerings that provide a subset of LAPACK's functionality. The first of these is a commercial library by EM Photonics called CULA Tools [31] and the second is a freely available library called MAGMA [16] developed by the same group responsible for the original LAPACK. Due to its open source nature and pedigree, MAGMA was chosen for this investigation.

In the LAPACK library, the LU decomposition of complex double precision matrices is performed using the ZGETRF routine. The matrices $[\mathbf{L}]$ and $[\mathbf{U}]$ obtained this way (as well as a vector of row pivots) are then used by the routine ZGETRS to perform the operations in (8) and (9) to solve the linear system. The performance of the MAGMA library on high-end CUDA capable devices such as the GTX 280 [25] outperforms a single core CPU implementation by a significant margin [20].

As discussed in [19] the limited memory on such devices means that out-of-core-like methods need to be investigated. Such a method based on the panel-based left-looking LU decomposition of [32] that makes use of CUBLAS as well as a hybrid approach that is built around MAGMA are presented by the authors in [19]. Although both of these overcome the memory limitations imposed by the device, their implementation contains a number of intricacies that are not suitable for an introductory text such as this. As such, only the use of the MAGMA and CUBLAS libraries to perform the LU decomposition is considered here. This is the same approach followed in [13].

Using this approach here serves two purposes. Firstly, it introduces the reader to the field of GPU computing using CUDA through third-party libraries, which hide a lot of the underlying detail of the implementation and as such provide a low cost (in terms of development time) means to harness the computational power of these parallel devices. Secondly, it provides a concrete example for the computational process as introduced in the previous section and shown in Figure 4. Although some of the details might differ (due to the abstractions provided by the libraries), the computational flow will be similar.

Consider the routine in Listing 1. It shows the code required to perform the LU decomposition of a matrix `Z` using MAGMA and CUBLAS. Note that the matrix is represented by a pointer of type `double2` (defined as part of CUDA) which is a `struct` with two `double` fields `x`, and `y`, with these used to store the real and imaginary component of a double precision complex value respectively. The other parameters, `N`, `LDZ`, and `IPIV`, represent the matrix dimension, the leading dimension of the matrix as stored in memory, and a pointer to a list to store the row pivots (calculated as part of the LU decomposition) respectively. The matrix (as is the case with most BLAS and LAPACK routines) is stored in column-major format with the leading dimension indicating spacing between the start of successive columns in memory [30].

Note that here the first step of import is to allocate the device memory for use in the computation. This is done by making a call to the **`cudaMalloc()`** routine which accepts (from first to last) the pointer where the address of the allocated memory

Listing 1. A routine to perform the LU decomposition of a matrix on a CUDA device using MAGMA and CUBLAS.

```
void LU_decomposition ( int N, double2* Z, int LDZ, int* IPIV )
{
  // determine the leading dimension of the MAGMA matrix must be a multiple of 32
  int LDM = N;
  if ( N % 32 != 0 )
    LDM = N + (32 - N % 32);
  // determine the internal MAGMA panel size
  int NB = magma_get_zgetrf_nb ( N );
  // allocate the memory on the device
  double2* pdev_Z;
  cudaMalloc ( (void**)&pdev_Z, N*LDM*sizeof(double2) );
  // MAGMA requires an additional workspace on the host
  float* host_workspace;
  cudaMallocHost ( &host_workspace, (NB*NR + 32*NB)*sizeof(double2) );
  // transfer the matrix Z to the device
  cublasSetMatrix ( N, N, sizeof(double2), Z, LDZ, pdev_Z, LDM );
  // perform the LU decomposition
  int INFO = 0;
  magma_zgetrf_gpu ( &N, &N, pdev_Z, &LDM, IPIV, host_workspace, &INFO );
  // check for an error in the LU decomposition
  if ( INFO != 0 )
  {
    printf ( "An error occured in the LU decomposition: %d\n", INFO );
  }
  // transfer the LU decomposition of Z back from the device
  cublasGetMatrix ( N, N, sizeof(double2), pdev_Z, LDM, Z, LDZ );
  // free the allocated memory
  cudaFreeHost ( host_workspace );
  cudaFree ( pdev_Z );
}
```

must be stored as well as the number of bytes to allocate as parameters [23]. Note that after the call to this routine, the pointer `pdev_Z` points to the start address of `N*NBM*16` contiguous bytes of memory allocated on the device. The memory allocated by **`cudaMalloc`**`()` must later be freed using a call to **`cudaFree`**`()` – which is equivalent to a `malloc()–free()` pair in standard C.

Although many of the MAGMA operations are performed in device memory, MAGMA also requires a work area on the host. This memory is allocated using the **`cudaMallocHost`**`()` CUDA routine. Once again, the routine has a pointer and a size (in bytes as parameters) although in this case, the pointer points to host memory and can thus be accessed safely in any host code that follows [23]. As is the case with the **`cudaMalloc`**`()` (and standard `malloc()`) routine, memory allocated this way must be freed – this time by the **`cudaFreeHost`**`()` routine.

As in Figure 4, following the allocation of memory on the device (and the host in this case), the input data needs to be transferred to the device. Although this can be done using a call to the **`cudaMemcpy`**`()` routine [23], the CUBLAS routine **`cublasSetMatrix`**`()` is used here instead [21]. The reason for this is that the CUBLAS routines wraps CUDA routines and thus provides a simpler interface for dealing with matrices, as is the case here. The first two parameters for the **`cublasSetMatrix`**`()` routine are the number of rows and columns in the matrix, followed by the size of each element. The final four parameters are a pointer to the source matrix (in host memory) and its leading dimension followed by a pointer to the destination matrix (in device memory) and its leading dimension.

Once the data is in device memory, the actual computation can begin. In this simplified case, this involves a call to the equivalent of the LAPACK ZGETRF (complex double precision LU decomposition [30]) routine as provided by MAGMA – aptly named **`magma_zgetrf_gpu`**`()`. The parameters to this routine are similar to a standard LAPACK implementation with the exception that a pointer to the workspace in host memory is included. It should be noted that the variant of the routine used here (with the `_gpu` suffix) leaves the computed result in device memory to be used later. Another implementation (**`magma_zgetrf`**`()`) is provided that transfers the input matrix to and from the device internally, although device memory and a host workspace still need to be allocated and freed by the user [20].

The final step in the process (apart from the freeing of the allocated memory already mentioned) is to transfer the output data back from the device to the host. This is done using the **`cublasGetMatrix`**`()` routine. This routine has the same format as **`cublasSetMatrix`**`()`, although in this case the source matrix (the fourth parameter) points to device memory and the destination matrix (the second to last parameter) points to host memory [21]. Once this is completed the matrix `Z` in host memory has be overwritten by the It is also possible to leave the LU decomposition of the matrix in GPU memory and use it in subsequent computations, but this is not considered here.

IV. Implementation: Assembling the impedance matrix

For the GPU acceleration of the matrix solution process, an off-the-shelf approach was followed. This is made possible by the fact that the linear algebra methods employed are common to a wide range of scientific and engineering disciplines and as such have received a lot of attention from the GPU programming community including NVIDIA themselves.

In the case of the method of moments matrix assembly, this approach cannot be used and it is required that we develop the lower level GPU code ourselves. In the context of this paper, this is an ideal case, as it allows for us to illustrate some of the finer points in GPU computing with the aid of our simple example. What is heartening is similar works such as those found in [6], [9], [8] were able to attain meaningful speed ups over existing CPU implementation. As discussed, each of these implementations differs slightly from our implementation and desired application and as such cannot be used verbatim.

Note that the discussion of this implementation does not make mention of the impact of the various CUDA memories on the performance [33]. The aim of this is to keep the implementation as simple as possible. Furthermore, since the individual matrix elements of (4) are independent of each other, the expected gain from using shared memory to share data between CUDA threads is negligible.

A. The development process

In many cases today, GPU acceleration is added to an existing implementation. Even if this is not the case, it is often easier to implement a CPU version of a given code first and then move on to a GPU implementation. This serves to provide reference values which greatly aid in debugging, as well as providing a CPU implementation for (if somewhat crude) performance comparisons.

The approach that we followed was to start with a Python implementation of the method as outlined in [14]. Python has a number of strengths making it an ideal language for rapidly developing a prototype code. Even though experience shows that the absolute performance of the Python code is significantly lower than that of a compiled implementation such as FORTRAN or C/C++, it still allows for an analysis of the relative time required by the various phases in the solution process. One of the great advantages of Python is its ability to be combined with native C/C++ or FORTRAN libraries using modules such as ctypes [34]. This makes it particularly suited to benchmarking.

As mentioned, a Python implementation's performance may be somewhat suboptimal resulting in biased performance comparisons with GPU implementations. For this reason the matrix assembly phase of the method of moments process is first reimplemented in C++ with this resulting implementation used as the CPU-based reference implementation for performance measurements. A third, GPU-based, version is then implemented using NVIDIA CUDA.

The use of the C++ and CUDA combination allows for the sharing of a lot of the code between the CPU and GPU-based implementation greatly decreasing the implementation as well as testing and debugging time. In theory this would also be possible using FORTRAN, but the FORTRAN compilers [35] for CUDA are commercial products and are not investigated further at present. The sharing of code between the CPU and GPU implementations is discussed further in subsequent sections and is an important point to consider in large software projects where regular maintenance and feature extensions are commonplace.

B. The computational process

The block diagram in Figure 5 serves to provide a better understanding of the computational process associated with the matrix assembly process. It also ties the equations from [14] in Section II-B1 to specific pieces of code and better illustrates how they are related. Evident in the figure that all the computations required to calculate a single matrix element (`Z[m,n]` $= Z_{mn}$ in (4)) can be grouped into a single computational unit or kernel which includes the evaluation of the integrals in (5) and (6). It should be noted that each call to the kernel is responsible for calculating a single matrix element and that a kernel need not be overly simple and can contain calls to other functions.

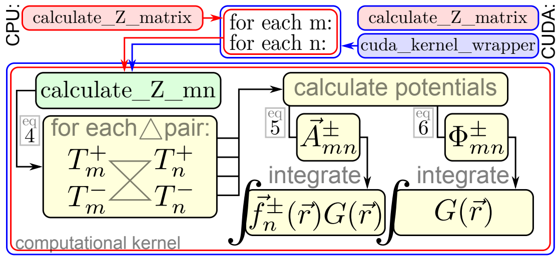

Fig. 5. A block diagram showing the computational process of the method of moment matrix assembly phase for both the CPU and CUDA implementations. The execution path for the CPU implementation is shown in red, with the GPU execution path shown in blue. Common elements are outlined in black and make up the shared computational kernel. The relevant equations from Section II-B1 for various sections are also indicated.

In [14] matrix assembly by faces (and not edges) is suggested as this allows for the reuse of computed integrals which improves performance. The face-based approach is not considered here due to the communication that would be required between the GPU threads making the implementation more complex, but not impossible. (For the problem considered here the face-based implementation was found to be about four times faster than the edge-based variant when considering the CPU implementation where communication is not such an issue). In the edge-based implementation there is no relationship between the matrix elements and thus no synchronisation needs to be performed between the kernels. This makes it a data-parallel task for which GPUs are well suited.

C. Problem domain segmentation

One of the first steps in the development of a GPU program (or parallel program in general) is to give some thought to the segmentation of the problem domain. As mentioned briefly in Section II-C, one of the strengths of CUDA is its ability to handle a large number of lightweight threads and that each of these threads can be identified uniquely and thus could be used to perform a calculation on a separate data element. Furthermore, these threads can be grouped into one, two, or three-dimensional blocks, with the blocks then arranged in a one or two-dimensional grid [23].

Since we are considering the assembly of the impedance matrix, which is a two-dimensional structure with each data element a matrix entry, intuitively it makes sense to make use of two-dimensional blocks as well as a two-dimensional grid. This is illustrated graphically in Figure 6, with a 5×5 grid of 8×8 blocks shown. The darker grey areas on the right and the bottom of the grid indicate portions of blocks that do not correspond with matrix entries. This is as a result of the matrix not being a multiple of the block size which must be the same for all blocks (although not necessarily the same in each dimension).

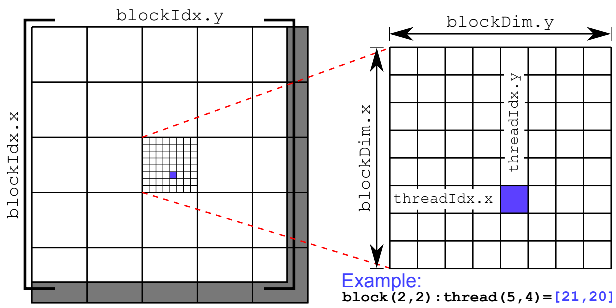

Fig. 6. A diagram depicting the arrangement of threads into two dimensional blocks and blocks into a two dimensional grid as used in this paper. Also shown are the CUDA variables used to determine the identity of a thread at runtime as well as an example thread with global index $[21, 20]$ highlighted in blue.

In CUDA the global row and column index of a given thread, `m`, and `n` respectively, can be computed from the thread's position in a block (`threadIdx`), the block's position in the grid (`blockIdx`), and the dimension of the blocks in the grid (`blockDim`) as follows:

```
int m = blockIdx.x * blockDim.x + threadIdx.x;
int n = blockIdx.y * blockDim.y + threadIdx.y;
```

The variables `threadIdx`, `blockIdx`, and `blockDim` are built-in multidimensional variables in CUDA of type `dim3` defined as:

```
struct dim3
{
  int x, y, z;
};
typedef dim3 dim3;
```

These are initialised at the time of the CUDA kernel call and will be discussed shortly.

D. Implementation details

With the problem domain segmentation discussed, what remains now is the discussion of how this is mapped to a CPU-based or GPU-based implementation. To aid this, we revisit the diagram of the computational process block diagram in Figure 5. It is clear that the process consists of two nested for loops over the observation and source edges related to the rows and columns of the impedance matrix respectively. The listing showing this for the CPU implementation in the function `calculate_Z_matrix()` is shown in Listing 2.

Listing 2. CPU implementation the function `calculate_Z_matrix` shown in Figure 5. Some parameters (mesh and geometry data) have been replaced with `"..."` for readability.

```
void calculate_Z_matrix ( int N, double *Z, int LDZ, ... ) {
  int m, n;
  double2* pd_Z = (double2*) Z;
  for ( n = 0; n < N; ++n ) {
    for ( m = 0; m < N; ++m ) {
      pd_Z[n*LDZ + m] = calculate_Z_mn ( m, n, ... );
    }
  }
}
```

Note that most of the parameters, such as the mesh data, have been excluded to improve readability, with only a pointer to the impedance matrix, `*Z`, and the number of degrees of freedom, `N` being shown. In each iteration of the loop, the `calculate_Z_mn ()` function is called which calculates the specified element of the matrix, Z_{mn}, and is also indicated in the block diagram in Figure 5.

The equivalent CUDA function is shown in Listing 3. Note that although the function name and the parameters are the same, the structure of the function is quite different. Most notable is the absence of any loops over the variables `m` and `n` as well as a missing call to `calculate_Z_mn()`.

Listing 3. CUDA implementation of the function `calculate_Z_matrix` shown in Figure 5. Some parameters (mesh and geometry data) and code have been replaced with `"..."` for readability. Where code has been omitted, comments are used to indicate the tasks that are performed.

```
void calculate_Z_matrix ( int N, double *Z, int LDZ, ... ) {
  // allocate and transfer geometric data to the GPU
  ...
  // allocate memory for Z on GPU
  double2 *pdev_Z = 0;
  cudaMalloc ( (void**)&pdev_Z, N*LDZ*sizeof(double2) );
  // determine the block and grid size
  int block_size = 8;
  int grid_size = 0;
  if ( N % block_size == 0 )
    grid_size = N/block_size;
  else
    grid_size = N/block_size + 1;

  dim3 grid = dim3(grid_size, grid_size);
  dim3 block =  dim3(block_size, block_size);
  // call the kernel wrapper
  cuda_kernel_wrapper <<< grid, block >>>
      ( N, pdev_Z, LDZ, ... )
  // transfer the result back from the GPU
  cudaMemcpy ( Z, pdev_Z, cudaMemcpyDeviceToHost );
  // free allocated GPU memory
  cudaFree ( pdev_Z );
  ...
}
```

What the CUDA version does contain is all the steps (as in Figure 4) to allocate and transfer the input data to the GPU (indicated as comments), the transfer of the resultant matrix from the GPU (`pdev_Z`) to the host (`Z`), as well as the set up of the grid of blocks as discussed in Section IV-C. Following this set up, there is a call to a wrapper function `cuda_kernel_wrapper()`, for which the listing is given in Listing 4. This function is declared with the `__global__` CUDA qualifier indicating that it is a CUDA kernel function that is executed on the device. All code up until the point where this function is called is executed on the host. Note that a CUDA kernel must be of return type `void`.

Listing 4. The CUDA kernel (indicated by the `__global__` qualifier) that acts as a wrapper for the computational kernel `calculate_Z_mn` as shown in Figure 5. Some parameters (mesh and geometry data) have been replaced with `"..."` for readability.

```
__global__ void cuda_kernel_wrapper ( int N, double2 *pdev_Z, int LDZ, ... ) {
  // identify the current thread
  int m = blockIdx.x*blockDim.x + threadIdx.x;
  int n = blockIdx.y*blockDim.y + threadIdx.y;
  // check if the thread falls in the matrix
  if ( ( m < N ) && ( n < N ) ) {
    // calculate the matrix element
    pdev_Z[n*LDZ + m] = calculate_Z_mn ( m, n, ... );;
  }
}
```

The CUDA code then, still contains no explicit loops, but does now contain a call to `calculate_Z_mn()`. The CUDA driver and runtime subsystem uses the `<<< grid, block >>>` part of the device kernel call, to initialise the number and arrangement of threads requested, and launches the function `cuda_kernel_wrapper()` for each of these threads [23]. Inside the function, each thread's unique ID is used to determine the data element (`(m,n)`) for which it is responsible and each thread calls the function `calculate_Z_mn()`, calculating the desired matrix element. Checks are also in place to see if the thread falls in the grey areas indicated in Figure 6, with threads that fall within these areas simply performing no calculations.

The calls to the `calculate_Z_mn()` routine in the case of the CPU and GPU implementation are identical, except that all call by reference parameters, such as the address of the impedance matrix (`pdev_Z`), for the CUDA version must be pointers to device memory and have been allocated using a call to **`cudaMalloc`**`()` [23]. The convention used by the authors is that a device pointer contains the prefix `pdev_`, as is seen in Listing 4.

The code of the computational kernel `calculate_Z_mn()` – called in both Listing 2 and Listing 4 – is identical for the CPU and CUDA versions. An outline of this routine, implemented in the file `computational_kernel.h`, is shown in Listing 5 and includes other functions and definitions common to both the CPU and CUDA implementations. Note that the definition of `calculate_Z_mn()` has a `DEVICE_PREFIX` before it. This is a preprocessor macro that is set to either `__device__` or blank depending on whether the code is do be compiled for a CUDA device or a CPU respectively.

Listing 5. An outline of the `computational_kernel.h` file containing the implementation of the calculation of the matrix elements `Z[m,n]`. Some parameters (mesh and geometry data) and code have been replaced with `"..."` for readability. Where code has been omitted, comments are used to indicate the tasks that are performed.

```
// include the common definitions
#include "common_defines.h"

// more computational functions corresponding to the blocks of Figure 5
...
// the implementation of calculate_Z_mn
DEVICE_PREFIX double2 calculate_Z_mn ( int m, int n, ... )
{
    double2 Z_mn;
    // calculate the value of the matrix element
    ...
    // return the calculated value
    return Z_mn;
}
```

In CUDA compilation, the `__device__` qualifier is used to specify that a routine (or variable) is to be compiled to be executed on a CUDA device [23]. This is similar to the `__global__` qualifier already discussed, except that a `__device__` function is not a CUDA kernel. An important point to remember is that only `__device__` functions can be called from within a `__device__` function and as such all the routines for the MOM matrix assembly implemented in `computational_kernel.h` must be preceded by the `DEVICE_PREFIX` macro.

A framework for primary file for the CUDA implementation, `calculate_Z_CUDA.cu`, is shown in Listing 6. In addition to the routines already shown in Listing 3 and Listing 4, the file also includes the definition of the `DEVICE_PREFIX` macro as `__device__`. Note that this occurs before the inclusion of the computational kernel (`computational_kernel.h`) and as such

TABLE I
THE FILES DISCUSSED IN THIS SECTION. THESE ARE REQUIRED FOR EITHER THE CPU OR THE CUDA IMPLEMENTATIONS (OR SHARED BETWEEN THE IMPLEMENTATIONS). NOTE THAT THE FILES, `COMMON_DEFINES.H`, `CPU_DEFINES.H`, AND `CUDA_DEFINES.H` HAVE NOT BEEN DISCUSSED IN DETAIL.

CPU	Shared	CUDA
`calculate_Z_CPU.cpp`	`computational_kernel.h`	`calculate_Z_CUDA.cu`
`cpu_defines.h`	`common_defines.h`	`cuda_defines.h`

the macro will be available as required for compilation. The inclusion of a file `cuda_defines.h` is also shown (although not discussed further) and contains any CUDA-specific definitions such as type and operator definitions.

Listing 6. An outline of the file `calculate_Z_CUDA.cu` showing the definition of the `DEVICE_PREFIX` macro used in Listing 5 (included as `computational_kernel.h`). Also shown is the inclusing of a header file containing CUDA-specific implementation details, `cuda_defines.h`. Some parameters (mesh and geometry data) have been replaced with `"..."` for readability.

```
#define DEVICE_PREFIX __device__
#include "cuda_defines.h"
#include "computational_kernel.h"

__global__ void cuda_kernel_wrapper ( int N, double2 *pdev_Z, int LDZ, ... ) {
  see Listing 4
}

void calculate_Z_matrix ( int N, double *Z, int LDZ, ... ) {
  see Listing 3
}
```

The framework for the CPU equivalent of `calculate_Z_CUDA.cu` is given in Listing 7 and is called `calculate_Z_CPU.cpp`. The file also shows the definition of `DEVICE_PREFIX` (this time as blank) as well as the inclusion of the file containing the implementation of the computational kernel. CPU-specific definitions are also realised using a separate header file (`cpu_defines.h`). Note that the declarations (names and parameter lists) for the CPU and CUDA implementations of `calculate_Z_matrix()` are identical as they are intended to be totally interchangeable.

Listing 7. An outline of the file `calculate_Z_CPU.cpp` showing the definition of the `DEVICE_PREFIX` macro used in Listing 5 (included as `computational_kernel.h`). Also shown is the inclusing of a header file containing CPU-specific implementation details, `cpu_defines.h`. Some parameters (mesh and geometry data) have been replaced with `"..."` for readability.

```
#define DEVICE_PREFIX
#include "cpu_defines.h"
#include "computational_kernel.h"

void calculate_Z_matrix ( int N, double *Z, int LDZ, ... ) {
  see Listing 2
}
```

A summary of the files mentioned in this discussion (and for which implementation they are required) is shown in Table I. Note that the specifics pertaining to the implementation of the MOM matrix assembly are restricted to the shared file `computational_kernel.h` while implementation related details, such as memory transfer and allocation in the case of CUDA, are contained in the device-specific files.

Although these implementations of the `calculate_Z_matrix()` routines can be used as part of a C/C++ implementation of the entire method of moments process, the approach considered here is to call them as part of a Python implementation. One of the ways to achieve this is to compile each of them as a shared library which can be loaded in Python using the ctypes module. The commands for compiling each of the implementations as a shared library under Linux are

```
g++ --shared -fPIC -o libcalculate_Z_CPU.so calculate_Z_CPU.cpp
nvcc --shared --compiler-options -fPIC -o libcalculate_Z_CUDA.so calculate_Z_CUDA.cu
```

with the `g++` compiler being used in the case of the CPU implementation and the NVIDIA C compiler `nvcc` (actually a compiler driver) for the CUDA implementation [23]. The compiler options are similar (`--shared` and `-fPIC`) and both generate a shared library file (`libcalculate_Z_CPU.so` and `libcalculate_Z_CUDA.so` respectively). These `.so` files can then loaded by ctypes and used in Python.

V. Results

There are two sets of results that are of interest here. The first are verification results where the implementations are compared to results presented in the literature as well as values computed with a commercial method of moments based code (FEKO [36]). The second set of results consist of a performance analysis of the method of moments process, where the focus is specifically on comparing the CPU implementation to the CUDA implementation.

For both the verification as well as the performance results, the same problem is considered, namely a square PEC plate with each side measuring one wavelength. The problem has already been introduced in Section II and a diagram representing the mesh used in verification results is shown in Figure 7. Also shown is the surface current distribution for a normally incident plane wave as computed using the CPU-based implementation.

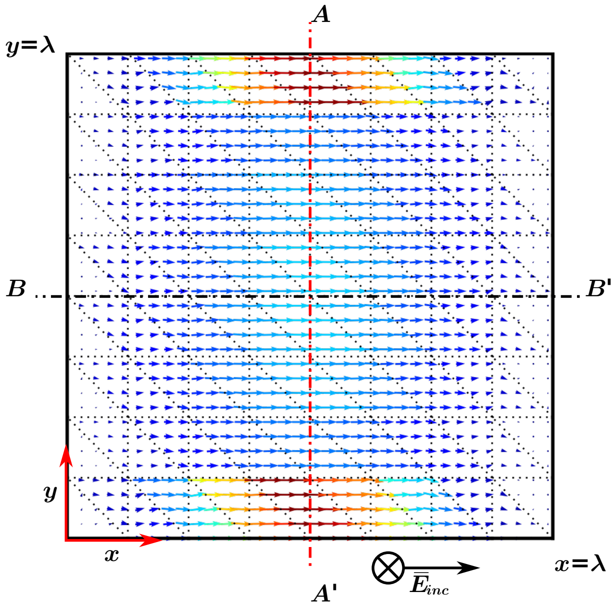

Fig. 7. A figure showing the 1λ square plate (lying in the xy-plane) used as a sample problem in this paper. The mesh (with 176 degrees of freedom) used to obtain the verification results as well as the cut-lines and incident field as defined in [14] are also shown. The surface current calculated using the CPU implementation is overlaid as a quiver plot.

A. Verification results

Since [14] was used as the starting point for the development of this code, the results from that paper are the first stop in verification of the results in this implementation. In Figure 6 from [14] the magnitude of the surface current on a square PEC plate illuminated by a normally incident ($\theta = 0$, $\phi = 0$) plane wave is evaluated along a vertical and a horizontal cut, AA' and BB' respectively. The cut lines as well as the polarisation of the incident field, $\vec{E}_{inc}$ are shown in Figure 7.

Figure 8 shows a recreation of figure from [14] with the addition of results calculated using FEKO [36] as well as the implementations considered here. For both the horizontal and vertical cuts the results obtained here agree well with both the FEKO results as well as the results from the original paper [14].

As a final verification step, the monostatic RCS of the same square plate is computed. Figure 9 shows the computed results for both the CPU and CUDA implementations compared to results obtained using FEKO. The CPU and CUDA results use the mesh as shown in Figure 7 which equates to 176 degrees of freedom, whereas the FEKO solution was computed using 261. The results computed using the methods presented here agree well with the FEKO results and the CPU and CUDA results are virtually indistinguishable from each other.

B. *Performance results*

For the performance analysis of the implementations presented here, only the matrix assembly, excitation vector calculation, and computation of the unknown current coefficients (LU decomposition and subsequent solution of the linear system) are considered – as discussed in Section II-B. Specifically, the relative performance of the CPU and CUDA implementations are considered for the matrix assembly and current coefficient calculation phases. The performance results are obtained on a 2.2GHz AMD Opteron 275 with 16GB of RAM, and a NVIDIA GTX 280 with 1GB of on-board memory for the CPU and CUDA results respectively. All timings are performed at the last common point between the CPU and the CUDA implementations, and thus include the time required to transfer data to and from the device in the case of the CUDA implementation. For the linear system solution phase, the single-core implementations of the AMD optimised BLAS and LAPACK routines provided by the AMD Core Maths Library (ACML) are used. It should be noted that in addition to the performance measurements for each of the CUDA runs, the relative error in the current coefficients was calculated and found to be to the order of numeric precision.

As an introduction to the relative importance of these phases consider Figure 10, which shows the percentage of the combined time required for each of the phases as a function of the number of degrees of freedom considered. From the graph it is clear that for the problems considered here (a square PEC plate with 64 to 7701 degrees of freedom) the matrix assembly phase accounts for the majority of the execution time. The solution of the linear system accounts for most of the rest of the execution time, although it does become more important as the problems get larger. This should further justify their selection as targets for GPU acceleration as presented here.

Since the matrix assembly phase is the dominant factor in the execution time, the acceleration thereof will offer the most total application speedup. If the CPU version of the matrix assembly routine is replaced with the CUDA version in the solution process the distribution of the execution times of the phases change as indicated in Figure 11. The CUDA implementation is clearly significantly faster as the relative contribution of the matrix assembly phase is now less than 15% for all the problem sizes considered. The effect of this acceleration in absolute terms is investigated at the end of this section.

An additional result of the acceleration of the matrix assembly phase is that the solution of the linear system as in (1) to find the unknown current coefficients now dominates the computational process. If the CUDA implementation of this (discussed in Section III) is also included, the relative distributions of execution time once again changes with the new distribution shown in Figure 12. Since the speedup of the solution of the linear system was not as great as in the case of the matrix assembly phase, the solution of the linear system still contributes the most significantly to the total execution time, and would be a good candidate for further acceleration.

Although the relative performance of the various methods is helpful in driving optimisation decision, one requires an absolute measure to properly evaluate the success of the accelerated implementation. To this end, the speedup of the CUDA

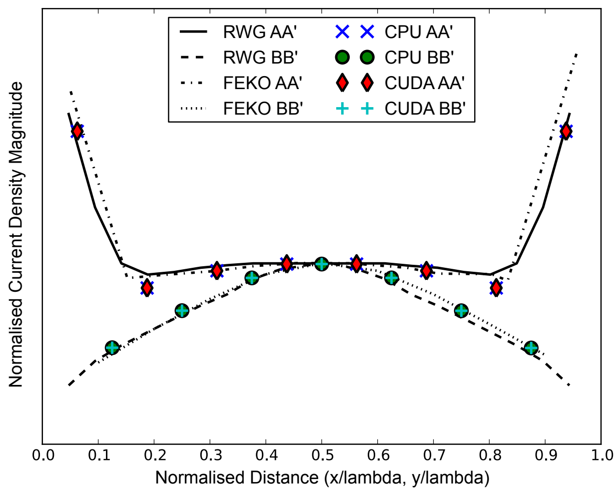

Fig. 8. A recreation of Figure 6 from [14] showing the normalised current density magnitude along the two cut lines in Figure 7. Results from [14] along with results computed using FEKO and the two implementation discussed here are presented.

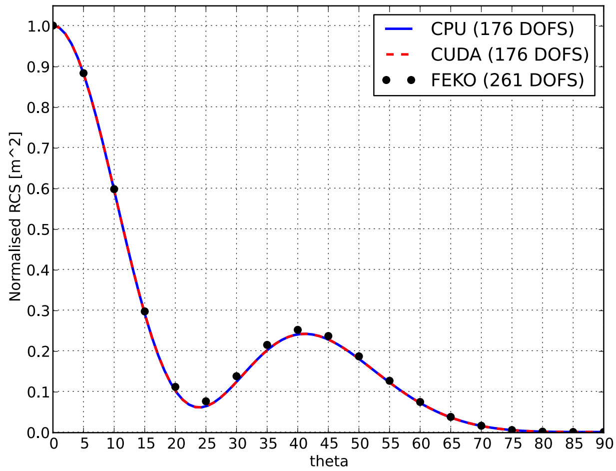

Fig. 9. The computed normalised radar cross section of the square plate shown in Figures 7 and 1 as a function of angle θ ($\phi = 0$). Results computed using FEKO as well as the CPU and CUDA implementations discussed here are shown.

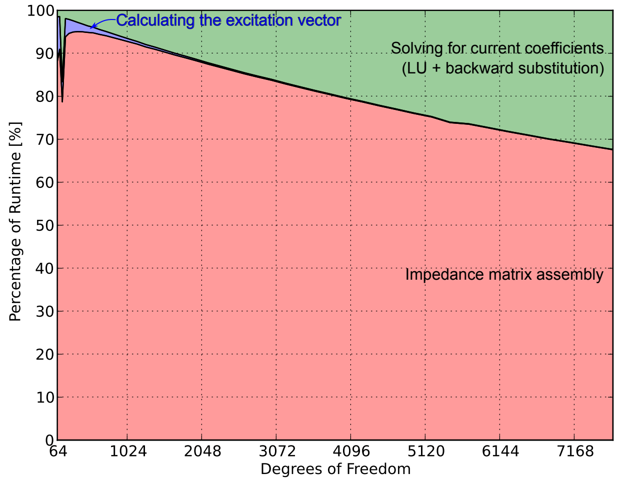

Fig. 10. The relative contribution of the three timed phases in the solution process as a function of the number of degrees of freedom in the mesh. The problem shown in Figure 7 is solved using the CPU implementation.

implementation over the CPU implementation, S_{CUDA}, is defined as

$$S_{CUDA} = \frac{t_{CPU}}{t_{CUDA}}, \tag{10}$$

with t_{CPU} and t_{CUDA} the execution time of the CPU and CUDA implementation respectively. Figure 13 show various speedup curves calculated using this definition. The matrix assembly (●) and system solve (■) curves in Figure 13 show the measured speedup for the individual phases of the solution process. The matrix assembly is about $150\times$ faster on the GPU than on the CPU with the solution of the linear system up to $12\times$ faster for the range of problem sizes considered. The speedup for the linear system solution phase agrees with the results published in [19] where random input data was considered.

The final two curves in Figure 13 show the total speedup measured for the combined runtime of the three phases considered here. The first of these (⧫) is the measured speedup when only the matrix assembly phase is replaced with the CUDA implementation (as in Figure 11). In this case the speedup is limited by them amount of time required for the linear system solution phase, thus the speedup drops as the contribution increases as shown in Figure 10. If the acceleration of the solution of the linear system is included, a speedup of about $40\times$ is attained as shown by the final curve (–).

VI. Conclusion

This paper has shown how the methods discussed in two prior works by the authors, [18] and [19], can be combined to provide a complete solution for the GPU acceleration of the method of moments in computational electromagnetics. In addition, more

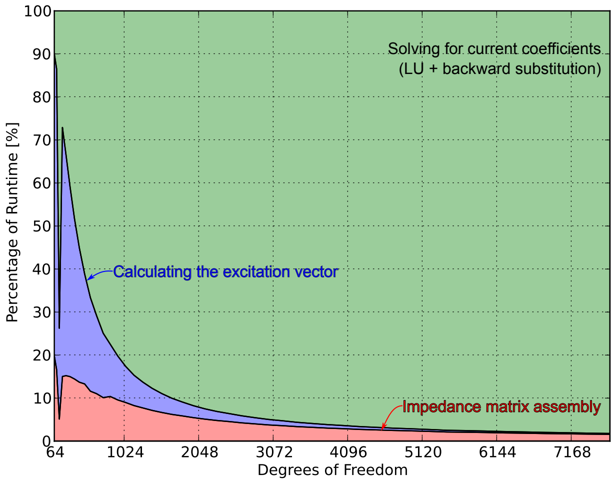

Fig. 11. The relative contribution of the three timed phases in the solution process as a function of the number of degrees of freedom in the mesh. The problem shown in Figure 7 is solved using a CPU implementation with the only matrix assembly phase performed using CUDA.

details on the implementation of these methods have been presented - with the aim of assisting readers interested in implementing their own GPU-enabled codes. The specific attention paid to the interaction, and more importantly code sharing, between the CPU and CUDA-based implementations of the matrix assembly phase of the method aims to provide further insight into how existing codes may be adapted for GPU computation.

The results shown further emphasise the fact that for a process such as the method of moments, it is not sufficient to consider each of the phases in the solution process independently - especially from a performance point of view. This has clearly been shown by the poor combined speedup performance if only the matrix assembly phase is accelerated. It was also found that the speedups achieved in each of the phases translate into a combined speedup of more than $40\times$ for complex double precision calculations, which greatly reduces the time required to obtain a solution.

One thing to point out is that the current implementation is limited by the amount of memory available on the GPU, and as such results are only shown for problems up to 7701 degrees of freedom. Although the CUDA-based LU decomposition as presented in [19] is implemented in such a way so as to overcome this limitation, the matrix assembly routine is not and this shortcoming is currently being addressed. The plan is to implement a blocking strategy similar to that of [9] for the matrix assembly phase and this is not expected to have too much of an impact on the performance of the phase.

Although the use of shared memory is not discussed explicitly in this paper, the impact on the performance of the matrix assembly routines is expected to be negligible as there is very little data that can be shared between the treads of a block. This

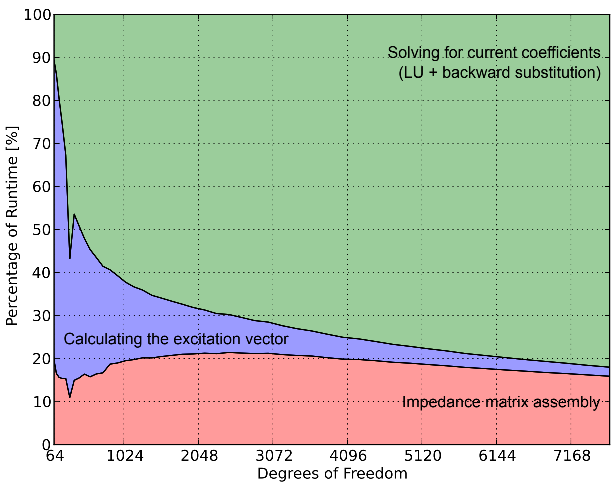

Fig. 12. The relative contribution of the three timed phases in the solution process as a function of the number of degrees of freedom in the mesh. The problem shown in Figure 7 is solved using the CUDA implementation.

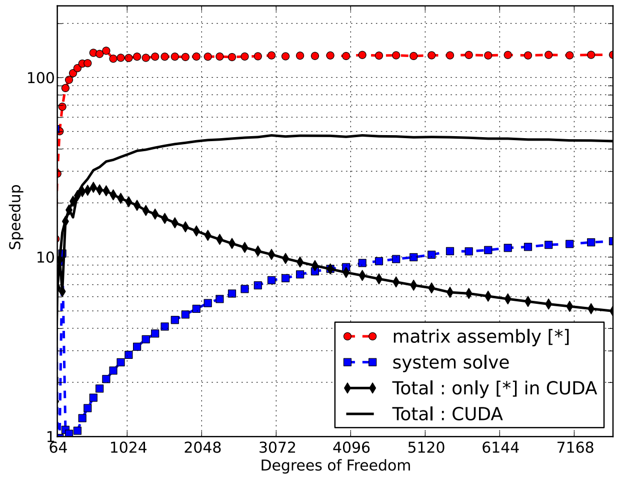

Fig. 13. The measured speedup of the matrix assembly (●) and solution of the linear system (■) as a function of the number of degrees of freedom. The combined speedup (–) as well as the speedup when only the matrix assembly phase is accelerated (♦) is also shown. A logarithmic scale is used on the vertical axis for better comparison.

is in contrast to the LU decomposition where shared memory is used to greatly improve the performance of matrix-matrix multiplication which forms an integral part of the process [26]. One way to improve performance may be to use automatically cached texture memory to store often used data such as mesh and geometric data.

In this paper, the CUDA acceleration of only two of the phases of the method of moments solution process have been addressed and timing results of a third phase, the calculation of the source vector, presented. Although this third phase was found to not contribute significantly to the total runtime of the system, the initialisation phase as well as the post processing phase may also account for a sizeable portion of the time required to obtain a solution. As such it may be fruitful to investigate their acceleration on GPUs.

Although CUDA is one of the foremost technologies in the field of GPU computing, there are other competing technologies available. One of the most interesting of these is OpenCL which promises compatibility with GPUs from various vendors as well as multi-core CPUs and other accelerating technologies. This is very attractive from a developer's point of view as it means that code can run on a wide range of systems without requiring additional development time. A continuation of this research may then be the adaptation of these methods to OpenCL, although this is encumbered by the lack of libraries such as MAGMA for this platform.

Even when only CUDA is considered, there are a number of points that can still be addressed. The first of these is the extension of these methods to make use of multiple CUDA devices installed in a given system. This is of particular importance in the high performance computing segment where units such as a Tesla S1070 [37] or S2070 [38] are often installed – with each providing four CUDA devices.

Acknowledgement

This research has been funded as part of a Flagship Project in conjunction with the Centre for High Performance Computing (CHPC) in South Africa. It has also been supported by funding from the National Research Foundation (NRF) of South Africa and EM Software & Systems – SA (Pty) Ltd.

The authors would also like to express their thanks to Danie Ludick for his assistance in developing the Python implementation of the matrix assembly routines as well as Willem Burger for his help in debugging the panel-based LU decomposition that forms the basis of the CUDA matrix solution routines.

References

[1] D. B. Davidson, *Computational Electromagnetics for RF and Microwave Engineers*, 2nd ed. Cambridge: Cambridge University Press, 2011.
[2] J. England, "A system for interactive modeling of physical curved surface objects," in *Proceedings of the 5th annual conference on Computer graphics and interactive techniques*. ACM New York, NY, USA, 1978, pp. 336–340.

[3] NVIDIA Corporation, "NVIDIA GeForce 8800 GPU architecture overview," Technical Brief TB-02787-001_v01, November 2006.
[4] ——, "CUDA Zone – The resource for CUDA developers," 2009. [Online]. Available: http://www.nvidia.com/cuda
[5] Advanced Micro Devices, "AMD Stream Computing," *http://ati.amd.com/technology/streamcomputing*, 2008. [Online]. Available: http://ati.amd.com/technology/streamcomputing/
[6] T. Takahashi and T. Hamada, "GPU-accelerated boundary element method for Helmholtz' equation in three dimensions," *International Journal for Numerical Methods in Engineering*, vol. 80, pp. 1295–1321, 2009.
[7] D. Bailey, "DSFUN90: Fortran-90 double-single package," *World-Wide Web site with software archives., March*, vol. 11, 2005.
[8] R. Chen, K. Xu, and J. Ding, "Acceleration of MoM Solver for Scattering Using Graphics Processing Units (GPUs)," in *Wireless Technology Conference*, Oriental Institute of Technology, Taipei, 2008.
[9] S. Peng and Z. Nie, "Acceleration of the Method of Moments Calculations by Using Graphics Processing Units," *IEEE Transactions on Antennas and Propagation*, vol. 56, no. 7, pp. 2130–2133, 2008.
[10] Khronos Group, "Opengl - the industry standard for high performance graphics," 2010. [Online]. Available: www.opengl.org
[11] Stanford University Graphics Lab, "Brookgpu," 2009. [Online]. Available: http://graphics.stanford.edu/projects/brookgpu/
[12] B. Virk, "Implementing method of moments on a GPGPU using nvidia CUDA," Masters Thesis, Georgia Institute of Technology, Atlanta, GA, May 2010. [Online]. Available: http://smartech.gatech.edu/handle/1853/33980
[13] D. De Donno, A. Esposito, G. Monti, and L. Tarricone, "Parallel efficient method of moments exploiting graphics processing units," *Microwave and Optical Technology Letters*, vol. 52, no. 11, pp. 2568–2572, August 2010.
[14] S. Rao, D. Wilton, and A. Glisson, "Electromagnetic scattering by surfaces of arbitrary shape," *IEEE Transactions on Antennas and Propagation*, vol. 30, no. 3, pp. 409–418, 1982.
[15] L. Tarricone, M. Mongiardo, and F. Cervelli, "A Quasi-One-Dimensional Integration Technique for the Analysis of Planar Microstrip Circuits via MPIE/MoM," *IEEE TRANSACTIONS ON MICROWAVE THEORY AND TECHNIQUES*, vol. 49, no. 3, p. 517, 2001.
[16] Inonovative Computing Laboratory, University Tennessee, Knoxville, "MAGMA: Matrix Algebra on GPU and Multicore Architectures," 2009. [Online]. Available: http://icl.cs.utk.edu/magma/index.html
[17] D. De Donno, A. Esposito, and L. C. L. Tarricone, "Introduction to GPU computing and CUDA programming: a case study on FDTD," *IEEE Antennas and Propagation Magazine*, vol. 53, no. 3, June 2010, in press.
[18] E. Lezar and D. Davidson, *GPU Acceleration of Method of Moments Matrix Assembly using Rao-Wilton-Glisson Basis Functions*. Kyoto, Japan: Proceedings of the 2010 International Conference on Electronics and Information Engineering (ICEIE 2010), August 2010, vol. 1, pp. 56–60.
[19] ——, "GPU-based LU Decomposition for Large Method of Moments Problems," *Electronics Letters*, vol. 46, no. 17, pp. 1194–1196, 19 August 2010. [Online]. Available: http://link.aip.org/link/?ELL/46/1194/1
[20] S. Tomov, R. Nath, P. Du, and J. Dongarra, "MAGMA Library," Tech. Rep. version 0.2, November 2009. [Online]. Available: http://icl.cs.utk.edu/magma/
[21] NVIDIA Corporation, "CUBLAS Library," Santa Clara, Tech. Rep. PG-00000-002_V3.0, February 2010.
[22] G. H. Golub and C. F. van Loan, *Matrix Computations*, 3rd ed. Baltimore: The John Hopkins University Press, 1996.
[23] NVIDIA Corporation, "Cuda programming guide," *NVIDIA: Santa Clara, CA*, 2008.
[24] Khronos Group, "OpenCL," 2009. [Online]. Available: http://www.khronos.org/opencl/
[25] NVIDIA Corporation, "NVIDIA GeForce GTX 200 GPU Architectural Overview," Technical Brief TB-04044-001_v01, May 2008.
[26] D. B. Kirk and W. W. Hwu, *Programming Massively Parallel Processors - A Hands-on Approach*. Burlington: Morgan Kaufmann, 2010.
[27] NVIDIA Corporation, "NVIDIA®OpenCL™JumpStart Guide," Technical Brief 1.0, February 2010.
[28] Advanced Micro Devices, "ATI Stream Computing OpenCL™," Programming Guide 1.03, June 2010.
[29] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Upper Saddle River, NJ: Addison Wesley, 2010.
[30] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*, 3rd ed. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1999.
[31] EMPhotonics, "CULA Tools - GPU-accelerated LAPACK," 2009. [Online]. Available: http://www.culatools.com/
[32] J. Dongarra, S. Hammarling, and D. W. Walker, "Key Concepts for Parallel Out-of-Core LU Factorization," *Computers and Mathematics with Applications*, vol. 35, no. 7, pp. 13–31, 1998.
[33] NVIDIA Corporation, "NVIDIA CUDA™: CUDA C Best Practices Guide," Tech. Rep. 3.2, August 2010.
[34] G. Kloss, "Automatic C Library Wrapping–ctypes from the Trenches," *The Python Papers*, vol. 3, no. 3, p. 5, 2008.
[35] The Portland Group, "CUDA Fortran: Programming Guide and Reference," The Portland Group, Tech. Rep. 102101747, August 2010. [Online]. Available: http://www.pgroup.com/doc/pgicudaforug.pdf
[36] EMSS, "Feko," 2010. [Online]. Available: www.feko.info
[37] NVIDIA Corporation, "Tesla s1070 gpu computing system," Specification SP-04154-001_v03, April 2010.
[38] ——, "Tesla 1u gpu computing systems," Specification SP-04975-001_v02, March 2010, preliminary Information.