Evaluation of Parallel Accelerators for High Performance Image Reconstruction for Magnetic Induction Tomography

Y. Maimaitijiang, M.A. Roula, S. Watson, G. Meriadec, K. Sobaihi and R.J. Williams

Abstract— **Magnetic Induction Tomography (MIT) is a new contactless imaging method for reconstructing the conductivity of objects. In MIT, one of main challenges is image reconstruction computation time, and the use of parallel processing is an effective means of reducing image reconstruction times to practical levels for monitoring applications. In this paper, we evaluated the comparative computational performance of three parallel processing accelerator options for MIT, namely (i) Graphics Processing Unit (GPU), (ii) Clearspeed AdvanceTM accelerator card, and (iii) multi-core processor PC, and discuss their advantages/disadvantages for application in MIT image reconstruction. The paper concentrates on parallelizing the finite difference (FD) algorithm, which is the most computationally demanding part of the forward model, and computation times for the implementation of this algorithm on each of the accelerators are given. The results show that the Clearspeed and quad-core accelerators provided similar performance displaying speed-up of 3.5 - 4 in comparison to a single processor implementation. The GPU accelerator however provided a substantially greater maximum speed-up of 70 using the same criteria. Given the high speed-up rates achieved, their relatively low cost and the availability of free development software tools, GPUs appear to be best suitable for acceleration of MIT imaging and monitoring.**

Index Terms— **Parallel processing, GPU, Clearspeed, multi-core processors, Magnetic Induction Tomography.**

I. Introduction

MAGNETIC Induction Tomography (MIT) is a contactless and non-invasive method for the imaging of the passive electrical properties, such as the conductivity distribution, within objects [1]. A set of excitation and detection coils is arranged around a sample (Fig. 1). The excitation coils are used to produce an alternating magnetic field, which then induces eddy currents within the sample. The detection coils detect the perturbation of the primary magnetic field produced by those eddy currents and the detected signal perturbations are then used to estimate the distribution of the samples electrical properties by solving an inverse image reconstruction problem. Potential medical and industrial applications of MIT include the detection and monitoring of cerebral strokes and the imaging of multi-phase flows such as process water in oil and gas pipelines [1,2].

The MIT forward model used for image reconstruction involves the solution of Maxwell's equations in three dimensions [3]. Typically, the solution must be iterated to address the non-linearity of MIT image reconstruction and the process is consequently very time-consuming. Furthermore, for medical applications the models employed require a high level of discretization which also results in long computation times. For instance, it takes typically over 10 minutes on a single processor workstation to carry out a single step reconstruction for an 80*80*80 voxel image in a 16-channel MIT system. However, for biomedical applications, such as cerebral stroke detection and monitoring, prompt image reconstruction is essential [2, 4]. It is therefore of paramount importance to reduce single-step computation times as much as possible to allow good quality images to be reconstructed in practical times. The Finite Difference (FD) algorithm we employ constitutes the most computationally intensive part of the forward model and contributes up to 90% of the total image reconstruction time. Accelerating the FD algorithm is therefore essential to achieve image reconstruction in practical time scales and parallel implementation of the FD computation is an obvious solution for addressing the computational limitation.

Various High Performance Computing (HPC) systems based on Multiple Instruction Multiple Data (MIMD) architectures have been applied in a broad range of medical applications including HPC assisted medical image analysis in surgery, 3D medical imaging and registration of medical imaging data [5,6]. In the related technique to MIT, Electrical Impedance Tomography (EIT), GRID computing has been applied to reduce image reconstruction times [7]. Accelerator techniques based on Single Instruction Multiple Data (SIMD) architecture have also increasingly been used in many imaging applications to reach a cost-effective solution, especially for those with medium computation requirements [8,9]. For example, in [9] 3D high resolution imaging for CT was accelerated with

Manuscript received January 9, 2011. This study was mainly funded by The Engineering and Physical Sciences Research Council (EPSRC) grant EP/f011059/1 under "HPC software development" call.

Y. Maimaitijiang is with Systems and Computer Engineering, Carleton University, 1125 Colonel By Drive, Ottawa, Ontario, K1S 5B6, Canada (E-mail: mamatjan@sce.carleton.ca).

M.A. Roula, S. Watson, G. Meriadec, K. Sobaihi and R.J. William are with the Faculty of Advanced Technology, University of Glamorgan, Pontypridd, CF37 1DL, Wales, UK

computation times reduced from weeks on a normal PC to hours using a system comprised of several GPUs.

Significant improvements in computation time have previously been achieved by our group through parallel implementation of the MIT forward model on an IBM SP supercomputer [10]. Large clusters and supercomputers however are expensive, and require very significant investments as regards space, power and maintenance. Another problem is queuing since such HPC resources are typically allocated among many HPC users due to their expense. This may not be appropriate for many medical and industrial applications which require dedicated resources and a prompt response.

Recent developments in parallel accelerator hardware have made it feasible to build HPC systems with computational performances comparable to clusters and supercomputers, but allowing small physical size, low power and low maintenance implementations. Previously, finite difference algorithms were demonstrated which took advantage of fast on-chip Graphics Processing Unit (GPU) shared memory to improve effective memory bandwidth and thereby increased performance [17, 18].

This paper presents the comparative performance results for implementation of a MIT FD algorithm on two currently available accelerator platforms, (i) an NVIDIA GPU and (ii) a Clearspeed Advance^{TM} accelerator card, and compares these with an implementation of the algorithm on (iii) an Intel multi-core PC.

The paper is organized as follows: first the FD algorithm will be described, details of the parallel implementation for the three proposed platforms will be given, test methodology will then be described and results given and discussed. The ultimate aim of this work is to produce a fast, discrete and cost effective iterative image reconstruction system for MIT.

II. Methodology

A. MIT systems

An MIT system is comprised of an array of excitation and detection coils. The coils are placed on coil formers which may be rigidly attached to a chassis or metal screen as shown in Fig. 1.

An image of electrical conductivity within a conductive sample is reconstructed by combining the measurement data from the MIT system with a sensitivity matrix **S**. The MIT forward problem is described by

$$\mathbf{S}\boldsymbol{\sigma} = \mathbf{b} \qquad (1)$$

where $\boldsymbol{\sigma}$ is the conductivity distribution vector and **b** is the measurement vector. Equation 1 is solved for the unknown $\boldsymbol{\sigma}$.

The sensitivity matrix describes the sensitivity of the received signal within each detection coil to variations in the value of

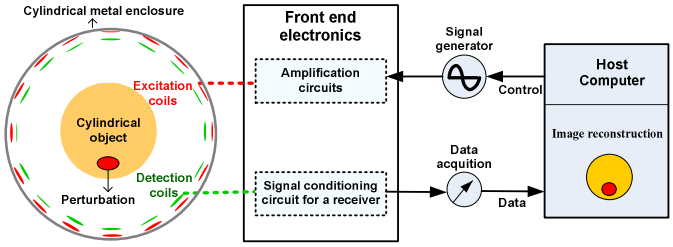

Fig. 1: A block diagram of the Glamorgan University MIT system

the conductivity within each voxel (a 3d pixel at positions (x, y, z)). **S** is computed using an electromagnetic model termed the forward model. The MIT forward model employed in this study is a quasi-static FD algorithm described in its single processor form in [3].

B. MIT Image reconstruction (FD algorithm)

The general procedure for creating the sensitivity matrix is shown below. Algorithmic details can be found in [10].

1) An analytically derived relation is used to compute the magnetic vector potential (**A**) produced by each excitation and detection coil (Fig. 1).
2) A finite-difference algorithm is then employed to calculate the electric scalar potential **Φ** using Kirchhoff's current law.
3) The induced eddy currents within the volume are then computed.
4) The sensitivity for each voxel and excitation/detection coil combination is then computed using modified reciprocity theorem [13] using (2).

$$\mathbf{S}_{i,j,k} = \sum_{i,j,k}^{L,M,N} \left(\frac{\mathbf{J}_{E_{i,j,k}} . \mathbf{J}_{D_{i,j,k}}}{\sigma_{i,j,k}} \right) + \frac{\mathbf{J}_{E_{i,j,k}} . \mathbf{J}_{D_{i,j,k}}}{\sigma_{i,j,k}} \qquad (2)$$

where $\mathbf{S}$ is the sensitivity matrix (also known as Jacobian), $\mathbf{J}_{E_{i,j,k}}$ and $\mathbf{J}_{D_{i,j,k}}$ are respectively the current density induced by the excitation and detection coils within voxel i,j,k, $\sigma_{i,j,k}$ is the conductivity of voxel i,j,k, and L, M, N are the total number of voxels along each direction.

The MIT forward model employed in this study is a quasi-static FD algorithm modified from [10]. The FD algorithm involves discretizing a volume into a finite cubic grid and approximating the derivatives [11]. The Jacobi method was used here to calculate the FD. The convergence rate of the Jacobi method is typically inferior to that obtained using for instance Successive Over-Relaxation, but has been adopted here due to the ease of its implementation for large 3D domains.

Kirchhoff's current law states that the sum of the six branch currents in voxel i,j,k results to zero. Scalar potentials $\Phi_{i,j,k}$ (Fig. 2) can be computed using the Jacobi method as follows,

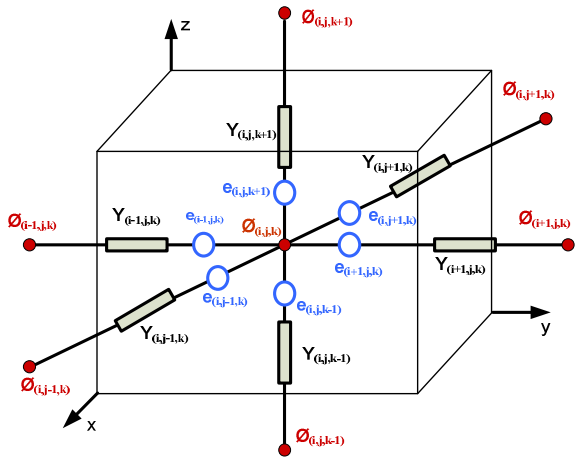

Fig. 2: Diagram showing electric scalar potential at each of, and the admittances Y and electromotive forces between, the centre of the cubic voxel element (i,j,k) and the centers of the 6 neighboring voxels.

$$\phi^{\eta+1}_{(i,j,k)} = \frac{Y_{(i-1,j,k)}(\phi^{\eta}_{(i-1,j,k)} + e_{(i-1,j,k)}) + Y_{(i+1,j,k)}(\phi^{\eta}_{(i+1,j,k)} - e_{(i+1,j,k)}) + Y_{(i,j-1,k)}(\phi^{\eta}_{(i,j-1,k)} + e_{(i,j-1,k)})}{(Y_{(i-1,j,k)} + Y_{(i+1,j,k)} + Y_{(i,j-1,k)} + Y_{(i,j+1,k)} + Y_{(i,j,k-1)} + Y_{(i,j,k+1)})} + \frac{Y_{(i,j+1,k)}(\phi^{\eta}_{(i,j+1,k)} - e_{(i,j+1,k)}) + Y_{(i,j,k-1)}(\phi^{\eta}_{(i,j,k-1)} + e_{(i,j,k-1)}) + Y_{(i,j,k-1)}(\phi^{\eta}_{(i,j,k+1)} - e_{(i,j,k+1)})}{(Y_{(i-1,j,k)} + Y_{(i+1,j,k)} + Y_{(i,j-1,k)} + Y_{(i,j+1,k)} + Y_{(i,j,k-1)} + Y_{(i,j,k+1)})} \quad (3)$$

where $Y_{(i,j,k)}$ is the impedance of the voxel, $\Phi_{(i,j,k)}$ is the scalar potential and $e_{(i,j,k)}$ is the magnetically-induced electric field strength modeled as a vector voltage generator, $\Phi^{\eta+1}$ is the newly calculated scalar potential in current iterations, Φ^{η} is the scalar potential from pervious iterations and η is the iteration step. The Jacobi algorithm runs on the distributed data iteratively using the old Φ^{η} to update with the new $\Phi^{\eta+1}$ in the memory after each iteration. The sequential FD algorithm can be written as a simple loop as shown in Fig. 3, where the first loop is the number of FD iterations.

```
for (iteration=0;iteration<nb_iteration; iteration++)
{
 for (k = 1; k < Z_SIZE-1; k++)
 for (j = 1; j < Y_SIZE-1; j++)
 for (i = 1; i < X_SIZE-1; i++)
       {
          Run  (Eq. 3);
       }
}
```

Fig. 3: Serial FD algorithm in C language.

C. Hardware specifications

1) Multi-core PC

Two multi-core PC's were employed utilizing a dual Xeon processor operating at 2.80 GHz with 3GB memory, and a quad-core Q9300 processor operating at 2.49GHz (6MB of L2Cache, S775 1333MHz) with 3GB memory.

2) GPU and CUDA

The GPU employed in this study was an Nvidia GeForce 8800 GTX Graphics card which was installed in a Dual Xeon PC. This card has 768 MB of onboard memory and 128 stream processors operating at 575 MHz clock speed and a memory bandwidth of up to 86.4 GB/s. More detailed specification of the GPU architecture can be found from [12]. For this study, the version 2.01 of NVIDA® CUDA™ [12] was used.

3) Clearspeed

The Clearspeed Advance board consists of two CSX600 processors. As shown in Fig. 4, the CSX600 consists of a mono execution unit (control unit), a poly execution unit with 96 processing elements (PE) and I/O units [13]. This architecture is based on Clearspeed's multi-threaded array processor (MTAP) core which is of Single Instruction Multiple Data (SIMD) type. The accelerator includes 1 GB of DRAM which mainly serves as a buffer for transferring data to and from a host PC. In CSX600, each PE has a very limited local memory of only 6 KB and a limited bandwidth of 3.2 Gb/s.

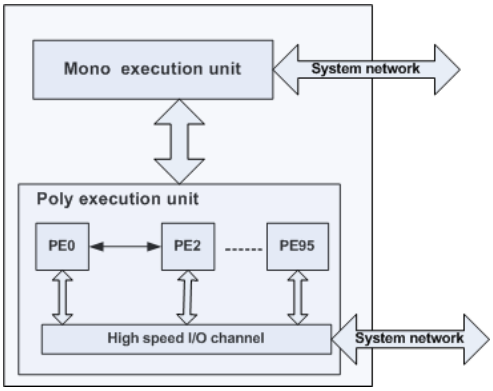

Fig. 4: An overview of the CSX600 processor architecture.

D. Parallel FD implementation

1) Parallel implementation of FD algorithm on Multi-core PC

OpenMP is a set of compiler directives that are used to instruct the compiler to produce programs that run in parallel on the shared memory processors of the individual nodes. Shared memory parallelization is achieved by inserting OpenMP directives which splits the Z loop range into parallel threads, allocated to one core each (Fig. 5).

The C++ code was compiled with a C++ compiler in Microsoft Visual Studio 2008 to run on the Multi-core PC. The number of shared processors was fixed (1 to N) before the test. Parallelization of shared memory systems is easier than for distributed machines due to the globally addressable space that multiple processors can access and share the same data. However, shared memory parallelization with OpenMP is a fork-join model of computation, which implies a synchronization point at each join operation. This leads to limited scalability of the system due to the heavy synchronization in the fork-join model [19]. A hybrid parallelization approach was therefore implemented by

combining Message Passing Interface (MPI) and OpenMP allowed using a huge number of processors, whilst improving the performance of FD computation on an IBM SP supercomputer [10].

```
for (iteration=0;iteration<nb_iteration; iteration++)
{
    #pragma omp parallel num_threads(nthreads) private(i,  j)
    {
      #pragma omp for
      for (k = 1; k < Z_SIZE-1; k++)
      for (j = 1; j < Y_SIZE-1; j++)
       for (i = 1; i < X_SIZE-1; i++)
         {
         Run  (Eq. 3);
         }
    }
}
```

Fig. 5: Pseudo code of FD implementation in multicore (#pragma stands for OpenMP directives).

2) Parallel implementation of FD algorithm on GPU

The GPU programming environment employed was Compute Unified Device Architecture (CUDA), released by NVIDIA [12]. CUDA is both a hardware and software architecture for issuing and managing computations on the GPU as a data-parallel computing device. CUDA provides an extension to the C programming language, called *kernels*, for source code development.

```
Copy (Admitance, Elecfield, Potetial)->GPU global memory
for (iteration=0;iteration<nb_iteration; iteration++)
{
   For each thread (Tx, Ty, Tz) and Block (Blckx, Blcky)
      Run
      {
            Potential  = 0
      Bx = Blckx
      By = Blcky *Modulo (NB_BLOCKS_Z)
      Bz = Blcky / NB_BLOCKS_Z

      i = Tx + Bx * BLOCK_SIZE_X
      j = Ty + By * BLOCK_SIZE_Y
      k = Tz + Bz * BLOCK_SIZE_Z
            Run  (Eq. 3);
              }
   Syncthreads
}
Copy (Potential)-> Host
```

Fig. 6: Pseudo code of FD implementation in GPU.

At the computation level, data are represented as a grid which is split according to 2D indexed blocks, each allocated to one multiprocessor. Each block is then divided into 3D or 2D parallel threads (Fig. 7).

The physical domain containing electric field vectors, admittances and scalar potentials for each voxel is divided into indexed blocks and allocated to each multi-processor within the GPU. The scalar potential value for each voxel is then computed on a separate thread. The result is gathered as a 2D grid and saved into the global memory. Fig. 6 shows the pseudo-code of the parallelization. CUDA handles most of the hardware details with the exception of the data mapping

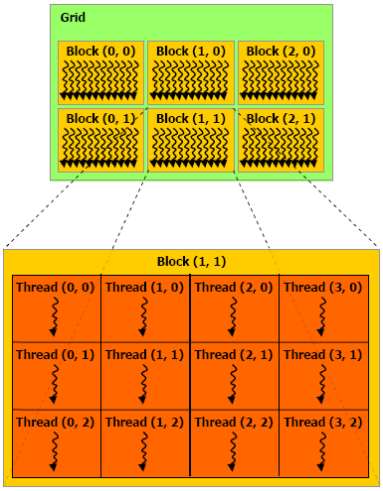

Fig. 7: Block-thread structure of GPU computational representation [20].

against block and thread indexes.

The result of our calculation is a 2D grid saved into the global memory. This grid is decomposed into 3D blocks of 8 voxels per dimension (so equal to the maximum size of 512 allowed), which are computed by each multiprocessor. Each multiprocessor run in parallel and whenever a new multiprocessor finish the process of the data, a new block of data is automatically computed by it.

3) Parallel implementation of FD algorithm on Clearspeed

Since the Clearspeed Advance board provides two CSX600 processors, a two-level parallelization approach was implemented based on MIT coils and physical domain to fully utilize the hardware resources and improve the performance. The first level of parallelization was achieved by dedicating one CSX600 processor to the excitation coils and the other to the detection coils. Excitation and detection coils are well parallelizable since the forward model is a "nearly embarrassingly parallel" problem when seen from the coil computation loops.

The second level of parallelization is achieved by splitting the domain between the 96 PEs on each CSX600 processor using the domain split approach described below. The FD algorithm was parallelized using Cn, a data-parallel extension to the C language for programming the CSX architecture.

A distributed parallelization with domain decomposition is a suitable approach on the Clearspeed accelerator. Here a volume is sliced into sub-domains and each PE computes the different sub-domains in parallel. In the domain decomposition approach, 3D matrices (Y, e, Φ) are decomposed into small sub-domains and distributed to all PEs to perform the FD computation. However, the PEs' have a limited autonomy without any branching unit. The mono execution unit is intended to control the program execution and to pass data to the PEs. The 96 PEs perform arithmetic operations in a SIMD manner. The FD algorithm based on the 3D domain decomposition requires the communication between neighboring nodes after each iteration to update the ghost values of the old Φ^{η} with the newly calculated $\Phi^{\eta+1}$, but only

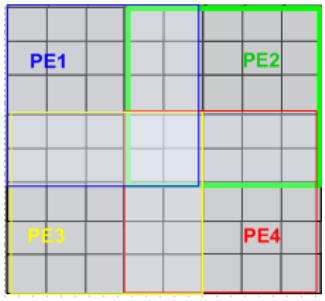

Fig. 8: 2D Illustration of pixel processing distribution amongst PEs

via the mono memory.

```
Copy (Admitance, Elecfield, Potetial)->GPU global memory

   //-- FD Iteration Loop --//
for (iteration=0;iteration<nb_iteration; iteration++)
{
  //--Copy and compute a subdomain in a round–robin manner--//
  PES= PE_NUM;
    memsetp(TMP_PE , 0, SIZE*sizeof(double));
    for (m = 0; m < USED_PES; m+= MAX_PES, PES+=MAX_PES)
    {
       if (PE_NUM < USED_PES-m)
       {
      //Copy data from a subdomain in card memory to PE memory
            for (j = 0; j < YPE_SIZE; j++)
            for (i = 0; i < XPE_SIZE; i++)
                  {
                        ADMITANCE -> ADMITANCE_PE
                        ELECFIELD  -> ELECFIELD_PE
                        PHI  -> PHI_PE
                        ID  -> ID_PE
                  }
   //--Perform FD computation ---//

        for (k = 1; k < ZPE_SIZE-1; k++)
        for (j = 1; j < YPE_SIZE-1; j++)
        for (i = 1; i < XPE_SIZE-1; i++)
         Run  (Eq. 3);
  //---Copy back the results (TMP_PE) to card memory---//
        for (j = 1; j < YPE_SIZE-1; j++)
        for (i = 1; i < XPE_SIZE-1; i++)
    TMP_PE -> PHI;
          }
     }
}
Copy (Potential) -> Host
```

Fig. 9: Parallel FD algorithm in Cn for Clearspeed.

CSAPI function calls were added to a C++ code on a host PC and the following steps were performed for the FD algorithm:

1. Moving the necessary data ((Y, e, Φ^{η}), and indices of a volume as shown in (2) from the host memory to the mono memory.
2. Moving the decomposed sub-domains from the mono to poly memory using the function memcpym2p.
3. Computing the FD algorithm in the poly memory.
4. Moving the newly calculated $\Phi^{\eta+1}$ from the poly to mono memory to update old Φ^{η} using the Clearspeed function *memcpyp2m*.
5. Repeating steps (ii), (iii) and (iv) until the convergence or maximum iterations are met the predefined values.
6. Moving $\Phi^{\eta+1}$ from the mono memory to the host memory.

Fig. 8 shows an object consisting of 8×8 voxels which includes predefined boundary voxels. The total number of unknowns is 7×7=49. Sub-domains of size 3×3 are transferred to the local memory of different PEs for parallel processing. In addition, the FD calculation requires the neighboring voxels to be transferred to the respective PEs, which results in allocating a sub-domain of 5×5 voxels as shown in Fig. 8.

The dimension of the sub-domains which may be stored and processed in the local PE memory is restricted by the size of this memory which is only 6KB for the CSX600 processor.

The above approach can be extended to the 3D case in a similar way. The corresponding sub-domains of the data (such as (Y, e, Φ^{η})) have to be loaded into the PEs in a round-robin manner in order to calculate the new $\Phi^{\eta+1}$ and the whole process has to be repeated for each iteration. The pseudo-code of the parallelization is shown in Fig. 9.

E. Simulation set-up and measurements

For all simulations a cylindrical conductive volume was discretized into cubic voxels varying in dimensions from 24*24*24 to 184*184*184 and the total computation times were measured for each case.

The speed-up SP_N for each accelerator method and domain size was defined as

$$SP_N = \frac{T_s}{T_N} \tag{3}$$

where T_S is the computation time for the algorithm on a single core CPU and T_N is the computation time for the algorithm on the specified device.

For the GPU, computation times were measured for FD computation within the GPU and data transfer time to and from host using the *cutCreateTimer* function in CUDA. For the multi-core workstation, computation time was measured using the *omp_get_wtime*() directives. The *GetTickCount* function was used for determining computation time for the Clearspeed measurements.

III. RESULTS

In this paper, we aimed to evaluate the performance of three parallel processing accelerators to select a suitable accelerator option for imaging and monitoring applications in MIT.

Table 1 shows the timing results for the FD algorithm on Quad Core PC (1 core), Dual Xeon (2 core), Quad Core PC (4 cores), Clearspeed and GPU with varying number of voxels.

In all cases the FD computation time increased with increasing domain sizes. The increase in computation time was not linear with the total number of voxels however since the number of FD iterations to achieve the specified level of

convergence also increased with domain size.

TABLE 1 FD computation times for varying number of voxels. (Time in seconds).

No. voxels	Quad Core (1 CORE)	Dual Xeon (2 CORE)	Quad Core (4 CORE)	Clearspeed	GPU
24^3	0.06	0.02	0.015	0.04	0.01
40^3	0.8	0.4	0.3	0.46	0.04
64^3	9.6	6.9	3.6	3.3	0.23
80^3	32	23.5	12.1	9.3	0,61
112^3	197	117	60	49	2.95
120^3	331	195	100	85	4.97
144^3	629	378	194	158	9.2
160^3	919	572	286	227	12.9
184^3	1749	1088	539	480	24.9

Table 2 shows the speedup factor obtained for 4 Core vs. 1 Core, ClearSpeed vs. 1 Core, GPU vs. Dual Xeon and GPU vs. 4 Core respectively for increasing number of voxels.

TABLE 2. Speedup factor for FD computation for varying number of voxels.

No. voxels	Speed up (4 Core vs. 1Core)	Speed up (ClearSpeed vs. 1Core)	Speed up (GPU vs. Dual Xeon)	Speed up (GPU vs. 4 Core)
24^3	4.2	1.7	4	1.5
40^3	2.7	1.9	10	7.8
64^3	2.6	2.9	30	16
80^3	2.6	3.4	38	20
112^3	3.3	4.0	40	21
120^3	3.3	3.9	39	20
144^3	3.2	4.0	41	21
160^3	3.2	4.0	44	22
184^3	3.3	3.7	44	22

The speedup factors obtained for the 2 core and 4 core implementations ranged from 1.6 – 3 and 3.2 – 4 respectively, close to the theoretical maximum values. Optimal speed-up was not achieved with this shared memory approach mainly due a heavy synchronization among processors. The speedup factors obtained for the Clearspeed were similar to the 2 and 4 core results ranging from 1.5 – 3.6 in comparison to a single processor implementation of the most time consuming FD algorithm.

The speedup obtained using the GPU was significantly higher than the other implementations ranging from 6 to 70 when compared to single core, from to 4 to 44 when compared to dual core and from 1.5 to 22 when compared to 4 core implementation. The speedup generally increased with increasing domain size.

IV. Discussion and conclusion

In this paper, MIT image reconstruction was implemented on three PC-based parallel processing platforms: GPU, Clearspeed and PC with multicores. We have attempted to select current, comparable and widely available models for this study. The results described in this paper would invariably change depending on the specific accelerator hardware used. However, we perceive this is unlikely to challenge the main conclusion given the significantly superior results obtained with the GPU.

For the multi-core PC implementations, efficiency dropped as the number of cores increased and optimal speedup was not achieved due to heavy synchronization among cores for the FD computation.

For the Clearspeed accelerator, the speed-up factor achieved was small. The performance profile on this card showed that the limitation on Clearspeed implementation was mainly the large amount of data transfers between mono and poly memories and the limited PE memory size (6KB), which resulted in high communication overheads. The communication time from the mono to poly memory took more than 96 % of the total time. The results achieved appear to agree with some previous published results [14, 15, 16] in that the Clearspeed Advance™ accelerator board appears to be most suited, or perhaps only suitable, for operations primarily involving matrix-matrix multiplications of large dense matrices of dimensions *m*n* where both m and n are large. Although multilevel parallel implementations, i.e. splitting the problem by both coil and physical domain, can in principle achieve higher speed up values, this requires the use of multiple Clearspeed Advance™ boards, which brings a very significant increase in cost.

The GPU provided, by far, the highest speedup. Its performance advantage over the Clearspeed is thought to be related to (i) the larger number of threads available, 12,288 versus 192 (PEs) between the GPU and Clearspeed respectively and (ii) the larger memory bandwidth available to the GPU with a significant bottleneck between the poly and mono memories existing in the Clearspeed (see Fig. 4).

The results showed that higher speedup was achieved for larger domain sizes, with larger problems resulting in better utilization ratios, and this is likely due to a decrease in idle time. For large scale problems, such as brain imaging with MIT for the detection of cerebral stroke, multi-level parallelization utilizing multiple GPU's should provide significant further reductions in image reconstruction times.

Based on the results obtained in this study, at present both Clearspeed accelerator and current dual and quad-core workstations appears to offer limited performance gains for the parallel implementations of the FD algorithm in comparison to the GPU implementation. The GPU implementation was also very cost-effective since GPU's are produced for mass consumer markets and the cost of suitable devices is therefore relatively modest (£100 - £500, 2010).

Given the high speed up rates achieved, the relatively low cost of GPUs, the availability of free development software tools, and the relative ease of development, GPU implementations appear to be very well suitable for acceleration of image reconstruction in MIT, and potentially in EIT and other tomographic methods.

ACKNOWLEDGMENT

This study was mainly funded by The Engineering and Physical Sciences Research Council (EPSRC) grant EP/f011059/1 under “HPC software development” call.

REFERENCES

[1] Griffiths H (2001) Magnetic induction tomography. Measurement Science & Technology, 12: 1126-1131

[2] Zolgharni, M., P. Ledger, D.W. Armitage, H. Griffiths, and D.S. Holder. "Detection of hemorrhagic cerebral stroke by magnetic induction tomography: FE and TLM numerical modeling". 2008 Electrical Impedance Tomography conference. 2008. Dartmouth, USA

[3] A. Morris, H. Griffiths, and W. Gough, "A numerical model for magnetic induction tomographic measurements in biological tissues," Physiological Measurement, vol. 22, pp. 113-119, 2001.

[4] M. Soleimani, "Computational Aspects of Low Frequency Electrical and Electromagnetic Tomography: A Review Study," Numerical Analysis and Modeling, vol. 5, 2008.

[5] Y. Dewaraja, M. Ljungberg, A. Majumdar, A. Bose, and K. F. Koral, "A Parallel Monte Carlo Code for Planar and SPECT Imaging: Implementation, Verification and Applications in I-131 SPECT," Journal of Computer Methods and Programs in Biomedicine, vol. 67, pp. 115-124, 2002.

[6] C. d. Alfonso, I. Blanquer, and V. Hernández, "HPC 3D Medical Imaging on Distributed Systems," Universidad Politécnica de Valencia (UPV), High Performance and Computing Group (GRyCAP) Dept. de Sistemas Informáticos y Computación (DSIC), Spain, 2003.

[7] J. Fritschy, L. Horesh, D.S. Holder, and R. H. Bayford, "Using the GRID to improve the computation speed of Electrical Impedance Tomography (EIT) reconstruction algorithms," Institute of Physics, vol. 26, pp. 209-262, 2005.

[8] K. Fatahalian, J. Sugerman, and P. Hanrahan, "Understanding the Effciency of GPU Algorithms for Matrix-Matrix Multiplication," Graphics Hardware, 2004.

[9] "Belgian researchers develop desktop supercomputer," The ASTRA research group, the University of Antwerp.

[10] Y. Maimaitijiang, M.A. Roula, S. Watson, R. Patz, R.J. Williams, H. Griffiths “Parallelization Methods for Implementation of Magnetic Induction Tomography Forward Models in Symmetric Multiprocessor Systems,” Journal of Parallel Computing 34 pp. 497-507, 2008.

[11] Y. Maimaitijiang, S. Watson, M.A. Roula, M. Zolgharni, H. Griffiths, R.J. Williams, “An Iterative Absolute Image Reconstruction Algorithm for Magnetic Induction Tomography”, Proceedings of 2008 Electrical Impedance Tomography Conference, Dartmouth, USA, June 2008.

[12] NVIDIA CUDA Homepage. Available at: http://developer.nvidia.com/object/cuda.html

[13] Clearspeed: Introduction to Clearspeed Acceleration. Clearspeed Technology Plc, 2008.

[14] V. Heuveline and J. P. Weib, “A Parallel Implementation of a Lattice Boltzmann Method on the Clearspeed Advance Accelerator Board,” University of Karlsruhe, 2003.

[15] Y. Yamamoto, T. Fukata, T. Uneyama, M. Takata, K. Kimura, M. Iwasaki, and Y. Nakamura, “Accelerating the Singular Value Decomposition of Rectangular Matrices with the CSX600 and the Integrable SVD,” Parallel Computing Technologies, 4671, 340-345, 2007.

[16] J. Bovay, B. Henderson, H. Lin, and K. Wadleigh, "Accelerators For High Performance Computing Investigation," High Performance Computing Division, Hewlett-Packard Company, 2007.

[17] P. Micikevicius. 2009. "3d Finite Difference Computation on GPUs Using CUDA," GPGPU- 2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units (New York, NY, USA), ACM, 2009, pp. 79-84.

[18] G. Meriadec, Y. Maimaitijiang, M. A. Roula, S. Watson, R. J. Williams “Acceleration of Finite Difference Algorithm on GPU for Application in Magnetic Induction Tomography,” EIT2009, Manchester, UK, 2009.

[19] G. Jost, H. Jin, D. a. Mey, and F. Hatay, "Comparing the OpenMP, MPI, and Hybrid Programming Paradigms on an SMP Cluster," NAS Technical Report NAS-03-019, 2003.

[20] CUDA Programming Guide, 2.1, NVIDIA. http://developer.download.nvidia.com/compute/cuda/2_1/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.1.pdf

[21] R. Amorim, G. Haase, M. Liebmann, and R. W. dos Santos. Comparing CUDA and OpenGL implementations for a Jacobi iteration. Technical Report SFB-Report No. 2008-025, Universität Graz, Graz, Austria, December 2008.