DOI 10.12694/scpe.v17i2.1156
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 17, Number 2, pp. 61–78. http://www.scpe.org

A GPU-BASED SOFT REAL-TIME SYSTEM FOR SIMULTANEOUS EEG PROCESSING AND VISUALIZATION

ZOLTAN JUHASZ AND GYORGY KOZMANN *

Abstract. EEG processing is generally acknowledged as a computationally very intensive task. The execution of pre-processing steps, frequency domain operations and source localisation algorithms result in long execution times, which prohibit the use of high-resolution EEG brain imaging techniques outside research laboratory settings. We present a novel GPU-based streaming architecture, which has the potential to drastically reduce execution times and, at the same time, provide simultaneous 2D and 3D visualization facilities. The system uses a highly-optimised and re-configurable pipeline of CPU and GPU cores that attempts to exploit the tremendous computing power whenever possible. The system can process live data arriving from an EEG device or data stored in EEG data files. The computer drives a large display wall system consisting of four 46-inch monitors, which provides a 4K-resolution drawing surface for visualising raw EEG data, potential maps and various 3D views of the patient's head. Two example brain imaging algorithms, the surface Laplacian and the spherical forward solution are used as an illustration for the effective use of the massively parallel GPU hardware in speeding up computations. The paper describes the architecture of the system, the key design decisions, and the performance optimization steps that were required to achieve sub-millisecond per-sample execution times. The control flow of the system is expressed in a very modular fashion in Java but the performance-critical algorithms are programmed in CUDA and run on the GPU. Relying on the CUDA-OpenGL interoperability bridge, the computing subsystem feeds visualisation results directly into the OpenGL pipeline, eliminating unnecessary GPU-Host data transfers. The system demonstrates that up to three orders of magnitude speedups are achievable compared to MATLAB implementations, and this processing speed can be maintained during simultaneous interactive 3D visualisation of the results.

Key words: Parallel Computing, GPU, EEG Processing, Brain Activity, Source Localisation

AMS subject classifications. 68Q10, 68W10, 92C55

1. Introduction.

High-resolution EEG imaging is an increasingly important tool in neuroscience [1]. Unlike CT or MRI that only provide structural information (head anatomy), EEG can measure functional brain activity. Its temporal resolution is typically in the millisecond range, which is vastly superior to alternative functional imaging methods, such as PET or fMRI. Consequently, EEG is indispensable in areas where brain activity mapping with high temporal resolution is required, e.g. in epilepsy diagnosis in the clinical setting [2, 3] or in cognitive experiments based on evoked or event-related potentials [4].

The neuroscience community relies on commercial and open source software products for EEG processing. Commercial programs are more common in the clinical area, whereas open source packages, such as the EEGLAB [5], Fieldtrip [6] or CSD [7] MATLAB toolboxes, dominate the research labs. While the open source approach reduces development time, facilitates sharing of methods and data, as well as serves as a verification infrastructure, by being de-facto standards they may halt innovation in the area of EEG software architecture and algorithm design. We argue that innovation is very much needed in these areas, since the processing speed achieved during routine evaluations with existing methods is intolerably low.

State-of-the-art EEG systems can use up to 256 electrodes and typically operate at a sampling frequency of 1-2 kHz . As a result, even a few-second measurement can generate hundreds of MByte's of data. Consequently, the evaluation of these experiments results in long execution times (varying from minutes to hours depending on the methods, data size and control parameters used in the experiments). This can be somewhat tolerated in single-shot experiments but it clearly presents a serious obstacle in large multi-patient studies, or if EEG is ever to be considered as a diagnostic tool in the daily clinical routine. Another complication is related to the visualisation of the results. Large amounts of intermediate results need to be stored in order to avoid repeated executions of long-running algorithms. Visualisation of temporal changes present special challenges too; the typical approach being to generate images of consecutive time samples and from these create a time-lapse video.

This paper presents a novel system developed in our department that employs a GPU-based massively parallel computational environment for simultaneous EEG processing and visualisation. The system demonstrates that near real-time processing speed is achievable in key brain imaging algorithms, such as the spherical surface

*Department of Electrical Engineering and Information Systems, University of Pannonia, Egyetem u. 10, Veszprem, Hungary, 8200 (juhasz,kozmann@virt.uni-pannon.hu).

Laplacian [8] or the forward solution for source localization [9], and shows that visualisation can be coupled to computation removing the need for storing intermediate data and/or generating time-lapse videos. While much work has been done in porting specific EEG algorithms to GPUs [10, 11, 12, 13], we are not aware of any system that are similar to ours in creating an end-to-end GPU-centric simultaneous compute and visualisation environment. Although our system is designed for EEG brain imaging, with little modifications, it can be readily used for ECG-based multi-channel body-surface potential mapping [14, 15, 16] applications as well.

The structure of our paper is the following. Section 2 introduces fundamental EEG imaging concepts and describes two illustrative methods, the (*i*) surface Laplacian map calculation and the (*ii*) EEG forward problem that is indispensable in brain activity source localisation. Both approaches aim at detecting the spatial and temporal patterns of cortical brain activity. The theory behind both methods is described briefly, along with key details of their traditional MATLAB implementations. Section 3 describes the rationale behind using graphical processors for increasing processing speed and discusses massively parallel GPU implementations of both algorithms. Section 4 describes the core architecture of our software and explains the key design concepts that resulted in a GPU-centric EEG processing and visualisation framework. Section 5 provides details of the visualisation subsystem and finally, Section 6 discusses our results.

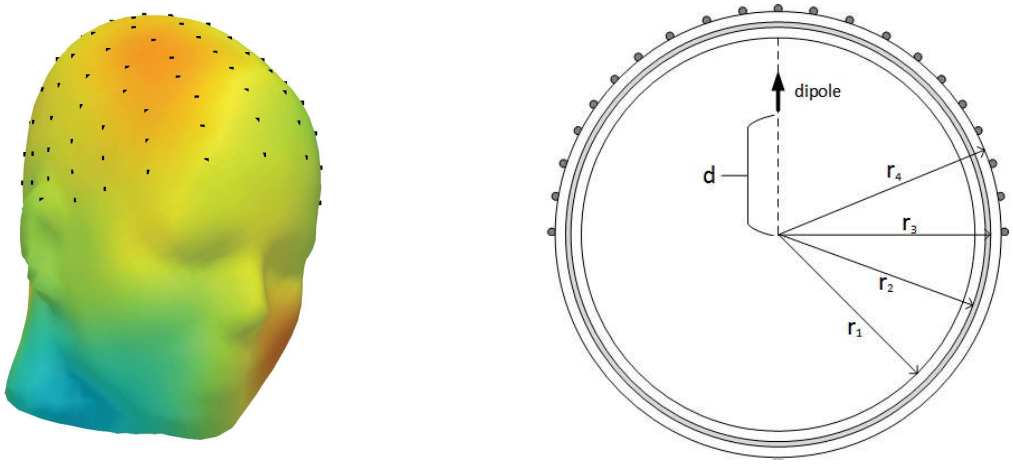
2. EEG imaging methods. EEG imaging is a cost-effective and non-invasive measurement method that maps the activity of the brain at very high temporal resolution. Its use is indispensable in epilepsy diagnosis and treatment, and in cognitive experiments that aim to understand the neural mechanism of various brain processes. The electric discharge of neurons generate a small but measurable potential distribution on the scalp surface, which is the basis of EEG brain imaging.

The electric potential distribution measured on the scalp is the effect of many simultaneously active neurons in the brain. These firing neurons are normally considered to be found in the cortical area and are represented as electrical current generators (dipoles) with a direction normal to the cortical surface. The electric field is largely influenced by the conductivity properties of the various parts (cerebrospinal fluid, skull, scalp) of the head. Most importantly, the relatively low conductivity of the skull results in a lateral spread of the potential distribution, which results in a "blurred" or "smeared" image on the scalp, as shown in Fig. 2.a. In addition, the measured signals have low signal-to-noise ratio and suffer from the presence of unwanted artefacts, such as the effect of eye and other muscle movements, skin resistivity changes, etc. As a consequence, the potential map is rarely used as recorded, but (i) first cleaned by performing additional filtering and other artefact removing pre-processing steps, then (ii) subsequent operations are employed that increase the spatial resolution of the EEG image and/or determine the location and amplitude of the current sources in the brain that generated the measured potential distribution. Two representative methods of the latter group are the calculation of the surface Laplacian and the source localisation based on the forward/inverse solution, which we describe briefly in the next sections. For practical reasons, these methods are often executed on a simplified, hypothetical 3 or 4-layer spherical head model, in which the brain, skull and scalp layers are represented by concentric spheres of varying radii, as illustrated in Fig. 2.b. While several methods exist that use realistic head models, in this paper we focus on the spherical head model only.

2.1. The surface Laplacian. Laplace imaging can be used to 'sharpen' the scalp potential images, i.e. to reduce the above-mentioned smearing effect. The surface Laplacian [17, 18, 19] – the second derivative of the surface potential – estimates the current source density (CSD) of the brain that is directly related to the activity level of cortical areas. The surface Laplacian is frequently used in event-related experiments, in which normally a multi-trial average of a 5-10 second time window of brain activity is analysed. The advantage of the Laplacian is that it is independent of the reference electrode choice, relatively insensitive to eye-movement artefacts and generates a relative topographic image that is well suited to the recognition of activity patterns. The Laplacian of the potential field f in the three-dimensional Cartesian coordinate system is expressed as

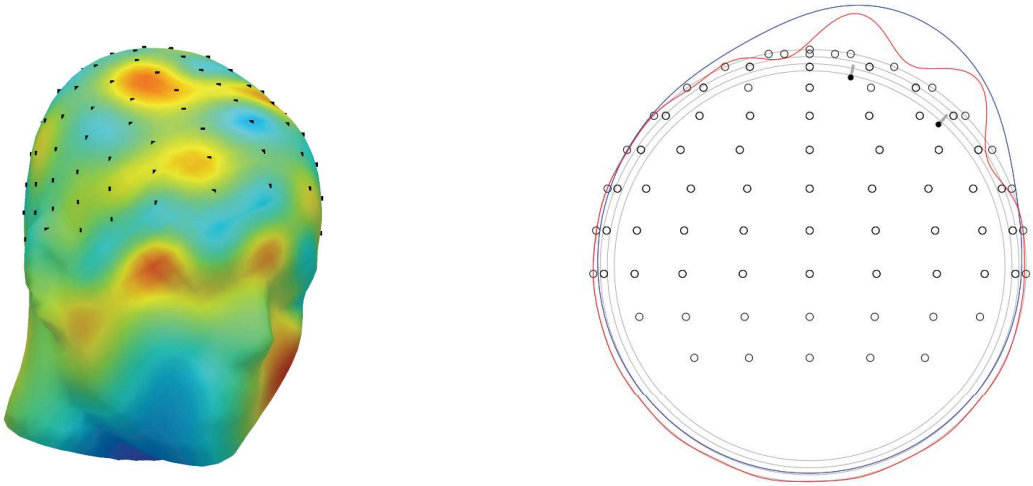
$$\Delta f = \nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} + \frac{\partial^2 f}{\partial z^2} \tag{2.1}$$

where f is the potential function represented by the discrete potential values measured at the electrodes. Fig. 2.2.a illustrates the power of the surface Laplacian in highlighting scalp current sources that are otherwise



(a) (b)

FIG. 2.1. *(a) EEG scalp potential map: A typical potential map displayed on a realistic head model along with the measuring electrodes. Red and blue colour represents positive and negative potential values, respectively. Face and neck areas should be ignored as they are the result of projecting values from the spherical interpolation. (b) 4-layer spherical head model: The spherical head model with a single radial activity source (electric dipole) with eccentricity d, and four structural layers (brain, inner and outer skull, scalp).*



(a) (b)

FIG. 2.2. *(a) The surface Laplacian of the scalp potential: Laplace map derived from the scalp potential distribution. (b) Difference in the activity source separation capability of the potential (blue) and Laplacian (red) maps: Comparison of the potential (blue line) and Laplace (red line) maps of a two radial dipole source configuration.*

invisible on the potential map of Fig. 2.a. Fig 2.2.b shows the focusing effect of the Laplacian on the cross section of the spherical model. The blue and red curves represent the potential and the Laplacian interpolated values, respectively. Although there are two active radial dipoles near the brain surface, the potential map does not indicate two sources, whereas the Laplace result clearly shows the presence of two dipoles.

One of the most widely used algorithm for computing the surface Laplacian on a sphere was proposed by Perrin *et al* [8, 20] that uses spherical splines for the calculation of the Laplacian. For a given surface point E

on the sphere, the surface Laplacian (a.k.a. the current source density), C, is given as

$$C(E) = \sum_{i=1}^{N} c_i h\big(cos(E, E_i)\big), \tag{2.2}$$

where E_i is the i^{th} electrode on the surface of the sphere and the c_i s are the solutions of the following system of equations:

$$\mathbf{GC} + \mathbf{T}c_0 = \mathbf{Z} \tag{2.3}$$

$$\mathbf{T'C} = 0 \tag{2.4}$$

where $T' = (1, 1, \ldots, 1)$, $C' = (c_1, c_2, \ldots, c_n)$, $Z' = (z_1, z_2, \ldots, z_n)$, and $G = (g_{ij}) = \Big(g\big(cos(E_i, E_j)\big)\Big)$, where z_i is the potential measured at electrode E_i. The potential and Laplacian interpolation functions $g(x)$ and $h(x)$ are defined as the sum of the following series

$$g(x) = \frac{1}{4\pi} \sum_{n=1}^{\infty} \frac{(2n+1)}{n^m (n+1)^m} P_n(x) \tag{2.5}$$

and

$$h(x) = -\frac{1}{4\pi} \sum_{n=1}^{\infty} \frac{-(2n+1)}{n^{m-1} (n+1)^{m-1}} P_n(x) \tag{2.6}$$

where $P_n(x)$ is the n^{th} degree Legendre polynomial and m is the stiffness parameter of the spline. The summation theoretically is infinite, in practice, it is sufficient to take only the first, typically $N = 15$ or $N = 50$ terms.

In the widely used MATLAB-based CSD Toolbox [7] package, the calculation of the surface Laplacian is carried out in two steps; first, the G and H matrices are computed using Eqs. 2.5 and 2.6, than the Laplacian is calculated for the entire *electrodes* $\times$ *samples* data window by solving Eq. 2.4. The outline of the MATLAB implementation is given below:

```
// high-level structure of the Laplacian (CSD) computation
M = electrode coordinate information;
[G,H] = GetGH(M);
D = interval of measured potential data;
D = D';
X = CSD (D, G, H);
```

where the functions getGH() and CSD() are defined as follows.

```
// calculation of the G, H matrices
function [G,H] = GetGH (M, m)
...
N = 50;                                  % set N iterations
G(nElec,nElec) = 0; H(nElec,nElec) = 0;  % claim memory for G- and H-matrices
for i = 1:nElec; for j = 1:nElec;
   P = zeros(N);                         % compute Legendre polynomial
   for n = 1:N;
      p = legendre(n,EF(i,j));
      P(n) = p(1);
   end;
```

```
   g = 0.0; h = 0.0;                    % compute h- and g-functions
   for n = 1:N;
      g = g + ( (( 2.0*n+1.0) * P(n)) / ((n*n+n)^m    ) );
      h = h + ( ((-2.0*n-1.0) * P(n)) / ((n*n+n)^(m-1)) );
   end;
   G(i,j) =  g / 4.0 / pi;              % finalize cell of G-matrix
   H(i,j) = -h / 4.0 / pi;              % finalize cell of H-matrix
end; end;

// calculation of the Laplacian
function [X, Y] = CSD(Data, G, H, lambda, head)
...
Gi = inv(G);                            % compute G inverse
for i = 1:size(Gi,1);                   % compute sums for each row
   TC(i) = sum(Gi(i,:));
end;
sgi = sum(TC);                          % compute sum total
fprintf('Sample points: %d\n', nPnts);
for p = 1:nPnts
   Cp = Gi * Z(:,p);                    % compute preliminary C vector
   c0 = sum(Cp) / sgi;                  % common constant across electrodes
   C = Cp - (c0 * TC');                 % compute final C vector
   for e = 1:nElec;                     % compute all CSDs ...
      X(e,p) = sum(C .* H(e,:)') / head; % ... and scale to head size
   end;
   if nargout > 1; for e = 1:nElec;     % if requested ...
      Y(e,p) = c0 + sum(C .* G(e,:)');  % ... compute all SPs
   end; end;
end;
```

2.2. The forward solution. While the surface Laplacian can help in focusing the scalp potential image, it does not provide information about the exact cortical location of the neuronal activity sources. Source localization methods can help to identify the location and magnitude of the activity sources (dipoles) based on the measured potentials. The difficulty lies in the problem that several dipole combinations can generate identical scalp potential maps. This is the bioelectrical *inverse problem* which has no unique solution except when a priori constraints can be applied.

The basis of the inverse solution is that one or more hypothetical dipoles are placed inside the brain and we compute the potential image generated by the given dipole — this is called the *forward problem* — which is then compared to the measured values. During this process, the position, orientation and magnitude of the dipoles are varied until an acceptable solution is found. Several solution methods have been proposed for solving the inverse problem; reviews and comparisons of these can be found in [21, 22, 23, 24].

The calculation of the forward solution — the potentials generated by a given dipole — is based on computing the electrical field generated by that dipole using a chosen head model. The simplest head model is the spherical model that can contain up to five layers. The advantage of potential calculations using the spherical head model is that they have a closed form solution, consequently their associated computational cost is relatively modest. This is crucial in the localisation process, during which a large number of dipoles need to be tested. Various algorithms have been developed for the multi-layer spherical head model forward calculation [25]. One of these is the 4-layer spherical forward solution algorithm proposed by Cuffin and Cohen [9] that is part of the Fieldtrip MATLAB toolbox [6]. The computation of the generated potential values is based on the following equation that calculates the effect of the x component of a source dipole vector. Similar equations are used for the y and z components.

$$V = \frac{P_x \cos(\varphi)}{4\pi\sigma_4 R^2} \sum_{n=1}^{\infty} \frac{(2n+1)^4 f^{(n-1)} (c*d)^{(2n+1)} P_n^1\big(\cos(\theta)\big)}{n\Gamma} \tag{2.7}$$

where P_n^1 is the associated Legendre polynomial,

$$\begin{aligned}\Gamma = {} & d^{(2n+1)}\big\{b^{(2n+1)} n(k_1-1)(k_2-1)(n+1) + \\ & c^{(2n+1)}(k_1 n+n+1)(k_2 n+n+1)\big\}\big\{(k_3 n+n+1)+(n+1)(k_3-1)d^{(2n+1)}\big\} + \\ & (n+1)c^{(2n+1)}\big\{b^{(2n+1)}(k_1-1)(k_2 n+k_2+n) + \\ & c^{(2n+1)}(k_1 n+n+1)(k_2-1)\big\}\big\{n(k_3-1)(k_3 n+k_3+n)d^{(2n+1)}\big\}\end{aligned} \tag{2.8}$$

and R is the radius of the scalp sphere, f is the coefficient for the dipole position (eccentricity), $k_1 = \sigma_1/\sigma_2$, $k_2 = \sigma_2/\sigma_3$, $k_3 = \sigma_3/\sigma_4$, whereas b, c, d are radius coefficients for each layer given as $b = r_1/r_4$, $c = r_2/r_4$, $d = r_3/r_4$, $R = r_4$.

```
// function for computing the forward solution
function [lf, vol] = eeg_leadfield4(R, elc, vol)
...
   % given a fixed volume conductor, these only need to be computed once for all electrodes
   const = (2*n+1).^4.*f.^(n-1) ./ (vol.t.*4*pi*c4*r4^2);
   % note that variable vol includes the Gamma term
   for i=1:Nchans
      % convert the position of the electrodes to spherical coordinates
      [phi, el] = cart2sph(elc(i,1), elc(i,2), elc(i,3));

      % change from colatitude to latitude and compute the cosine
      cos_theta = cos(pi/2-el);

      % the series summation starts at zero
      s_x = 0;
      s_z = 0;

      for n=1:Nmax
         P0  = plgndr(n,0,cos_theta);     % zero'th order Legendre
         P1  = plgndr(n,1,cos_theta);     % first order Legendre
         s_x = s_x + const(n)*P1/n;       % s_y is identical
         s_z = s_z + const(n)*P0;
       end

    lf(i,1) = -cos(phi) * s_x;
    lf(i,2) = -sin(phi) * s_x;            % s_y is identical to s_x
    lf(i,3) = 1         * s_z;
end
...
```

2.3. Computational cost. In this section, we list the execution times of the surface Laplacian and forward solution computation using the CSD Toolbox and Fieldtrip MATLAB implementations described in the brevious sections. These results serve as baseline performance indicators that highlight the performance

TABLE 2.1
Calculation time [sec] of the **G** *and* **H** *matrices (CSD Toolbox)*

n	Number of electrodes		
	64	128	256
15	5.74	22.84	92.50
50	34.00	137.51	552.12

TABLE 2.2
Total surface Laplacian calculation time [sec], N=15, in MATLAB (CSD Toolbox)

Number of samples	Number of electrodes		
	64	128	256
1	5.74	22.84	92.51
512	5.81	23.03	92.89
1024	5.89	23.15	93.28
2048	6.04	23.47	94.07
5120	6.50	24.40	96.33
7160	6.80	25.03	97.94

problems of traditional sequential implementations, and indicate what speedups are required for achieving online processing and visualisation. All values are obtained with an Intel Core i7-3820 2.70 GHz processor.

The CSD Toolbox surface Laplacian execution times are listed in Tables 2.1 and 2.2. Since this implementation requires an initialisation step, Table 2.1 lists the calculation time of the **G** and **H** matrices for different numbers of electrodes and summation limits, while Table 2.2 shows the overall execution time including the **G**, **H** matrix and the sample window Laplacian calculations for $N = 15$. It is evident — even at this summation limit, N — that the computation of the surface Laplacian for a single sample can take 10s of seconds. For $N = 50$, the execution time reaches the order of minutes. Note that in our measurement setup the sampling frequency is 2048 Hz, thus the evaluation of 1 second of EEG data requires the processing of 2048 samples.

Table 2.3 contains illustrative execution times obtained with the Fieldtrip Toolbox using 64 and 128 electrodes and varying number of dipoles. The forward solution is a truly critical operation since it has to be calculated many times during the solution of the inverse problem. In the most general case, the location, orientation and magnitude parameters of each dipole are adjusted iteratively until the best fit is found, therefore several iterations are required. Although a dipole count of 1,000 or 10,000 might seem large, in a spherical head model more than 5,000, in a realistic cortical model, more than 100,000 dipoles might be required in order to achieve sufficient spatial resolution.

3. Embracing GPU technology. The traditional approach to speeding up the illustrated MATLAB algorithms is to implement them in C/C++, potentially using such a parallel implementation technology that can effectively scale over a range of CPU cores. Unfortunately, in traditional computers the achievable parallelism – hence speedup – is limited by the number of CPU cores. In commodity personal computers, one cannot expect speedup values larger than 8. On HPC servers using state-of-the-art Xeon processors, the potential speedup can reach up to 16. Note that we ignore large multi-node supercomputers with hundreds or thousands of CPU cores as they are neither typical nor practical within the medical field. The execution times listed in Tables 2.1 to 2.3 show clearly that orders of magnitude larger speedup values are required if the target is to reach near real-time execution speed.

A rough estimate for the required computational performance can be made as follows. Let us assume a forward calculation problem for $E = 128$ electrodes, $D = 5200$ dipoles, $S = 2048$ samples/sec with $K = 1578$ operations for computing the effect of one dipole on one scalp electrode. The required number of operations per second is $E \times D \times S \times K = 1.28 \times 5.2 \times 2.048 \times 1.578 \times 1011 = 2.045 \times 1012$, which gives approximately 2 teraflops. A similar estimate for the surface Laplacian with $E = 128$ electrodes, $P = 5120$ surface points, $S = 2048$ samples/sec, and $K = 1200$ operations per surface point results in $1.28 \times 5.12 \times 2.048 \times 1.2 \times 1011 = 1.61 \times 1012$ operations, or 1.61 teraflops. This level of performance cannot be achieved with 8-10 CPU cores. Graphical processing units, however, provide a cost and energy-efficient high-performance computing platform for EEG processing.

TABLE 2.3
Calculation time [sec] of the 4-layer forward solution in MATLAB (Fieldtrip Toolbox)

Number of samples	Number of dipoles					
	1	10	100	500	1000	10000
64	0.25	0.52	3.34	15.93	31.90	314.86
128	0.32	0.95	7.22	35.13	70.03	700.02

3.1. GPU overview. Graphical processors, due to their origin in 3D graphics, are designed for massively parallel execution of vertex/pixel level instructions. Today's GPUs are general-purpose, programmable computing engines. State-of-the-art GPU chips can contain over 3000 cores and their computational performance can exceed 6 teraflops (using single-precision operations). GPUs can efficiently solve large data-parallel problems if the problem maps effectively to a large number of GPU cores and is based on executing the same instruction sequence for the data set. As illustrated in Figure 3.1, the GPU program (called kernel) is executed under the control of a host program. The kernel represents a thread of execution that is executed in multiple copies at the same time. Kernels access data from the GPU device and on-chip memory during execution, hence data must be transferred between the host system and the GPU before and after the kernel execution. Kernel threads can be arranged into 1D, 2D or 3D grids in order to efficiently map a problem onto the physical cores. Threads are executed in warps (groups of 32 threads) under the control of an on-chip thread scheduler. The GPU card uses a non-uniform access memory; the various layers of the memory hierarchy (registers, shared memory, constant and global memory) have very different access times. Register and shared memory operations are the fastest. Shared memory is also accessible to all threads within a block and can be used to communicate values among the threads. Global memory access is very expensive, thousands of instructions can be executed within a single read/write cycle. Even more expensive is the data transfer operation between the host computer and the GPU.

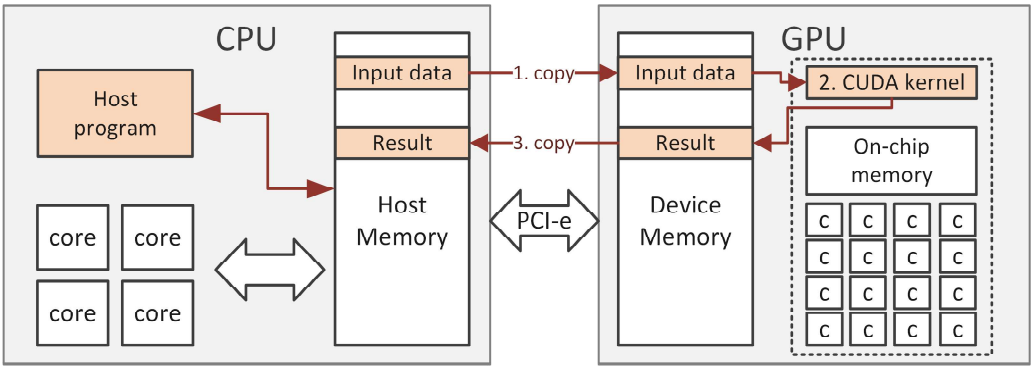


FIG. 3.1. *Illustration of the typical GPU kernel execution steps, including host–GPU data transfer.*

3.2. Parallel implementation strategy. Our primary aim in creating parallel GPU implementations for the Laplacian and the forward computation was to investigate various performance optimisation approaches and test how fast we can execute these algorithms on graphics cards. In order to have maximum flexibility and maximum control over the hardware during our implementation and optimisation steps, we have excluded OpenACC [26] and library-based (e.g. cuBLAS) approaches in our solution and developed the programs at a lower abstraction level using CUDA [27].

As the first step, we re-implemented both algorithms in C and in Java. They provided the foundation for the parallel implementations. Then, we created a data-parallel implementation of both the Laplacian and the forward algorithms. The goal of the first implementation was to numerically validate the parallel versions and to identify performance critical points. Finally, we experimented with various data partitioning schemes, memory and instruction scheduling optimisations to improve performance. Our aim was to create implementations that scale over a large number of cores, minimise memory transfer and the number of cycles not spent with computation.

3.2.1. The GPU surface Laplacian. In Perrin's original surface Laplacian algorithm and in its CSD Toolbox MATLAB implementation, the Laplacian is only computed for the electrode coordinates. The other points are linearly interpolated during plotting the results. In our system, we wanted to compute the values at every vertex position of the head as the output of the Laplacian kernel is directly fed into the OpenGL pipeline's vertex shader. Our implementation computes interpolated values for each surface point (vertex), hence our main loop is executed S times, where S typically varies between 5,120 and 15,000, depending on the head model mesh. Since each surface point calculation requires all 128 electrode coordinates, our first implementation was distributed into S independent threads, each computing one surface point value. The outline of the Laplace kernel is as follows.

```
__global__ void laplace_kernel(
   const int numPoints, const float *d_surface_x, const float *d_surface_y,
   const float *d_surface_z, const float *d_electrodes_x, const float *d_electrodes_y,
   const float *d_electrodes_z, const float *d_potential, float *d_laplacian, float* mX)
{
   int i = blockDim.x * blockIdx.x + threadIdx.x;
   ...
   float interpolVal = 0.;
   for (int j = 0; j < 128; ++j) {
      interpolVal += laplace_interpol_device(d_surface_x[i], d_surface_y[i],
      d_surface_z[i], d_electrodes_x[i], d_electrodes_y[i],
      d_electrodes_z[i]) * mX[j];
   }
   d_laplacian[i] = interpolVal / ((sphereRadius ) * (sphereRadius));
}
```

The function `laplace_interpol_device()` is our GPU device function executing one step of the Laplace interpolation calculation; it calculates the Legendre polynomial and the value of $h(x)$ based on 2.6.

In the second implementation version, the problem was partitioned into an S block by E thread grid, where S is the number of surface points and E is the number of electrodes. This partition enables us to compute the partial Laplacian terms in the series of Eq 2.2 in parallel ans store the results in a shared memory array that is available to all threads of a block. These partial results are then summed to provide the final result for each surface point by a reduce-sum operation, implemented using the `__shuffle_down()` CUDA function. This implementation further reduced the execution time of the Laplacian.

3.2.2. The GPU forward solver. The forward algorithm is similar in structure to the Laplacian algorithm, in that it computes the potential for each electrode from the combined effect of the dipoles placed in the head model. The effect of each dipole on one specific electrode can be evaluated independently from the others, but the final result must be computed as the superposition of the partial results, therefore requiring a reduce-sum operation. Consequently, we used a block-partitioning scheme for the dipoles $\times$ electrodes data domain.

Our first parallel forward solver implementation was based on the work reported in [28]. The underlying structure of the reported algorithm is a dipoles $\times$ electrodes thread grid, in which each thread calculates the effect of one dipole on one scalp electrode. We have added several performance optimizations to this algorithm. Using a more optimized grid allocation strategy, restructuring code to optimize memory access and exploiting on-chip instruction level parallelism, we were able to almost double the performance of the original algorithm.

Our next modification for improving the overall execution time was to use a more efficient baseline algorithm. The forward algorithm proposed by Sun [29] uses an interpolated function to eliminate the lengthy computation of the Legendre sums in the original Cuffin & Cohen algorithm. This algorithm reduces the number of floating point operations to 140 per steps, bringing another ten-fold increase in performance[29]. The outline of this kernel is as follows. Input parameters are the x,y,z coordinates of the dipole, x,y,z dipole moments, x,y,z

coordinates of the electrodes (here called sensors).

```
_global__ void kernel_potential_Sun(float *dipx, float *dipy, float *dipz,
   float *momx, float *momy, float *momz, float *senx, float *seny,
   float *senz, float *out, int dipole_num, int sensor_num)
{
    int dipole_id = blockIdx.x;
    int sensor_id = threadIdx.x;

   __shared__ float potentials[128];

   ...
   vr = cn[1]*x+cn[2]*f*(1.5f*x2 - 0.5f)+a[0]*(Q-1.0f) * rsqrt(p)
        +a[1]*(x-f)*Q3+a[2]*(f*(x2+f2-2.0f)+x*f12)*Q5+cn[3]*f2*x*(2.5f*x2-1.5f)
        +a[3]*(x+f*((5.0f*x2-4.0f)+f*((x*(x2-9.0f))+f*((10.0f-2.0f*x2)-fx-f2))))*Q7;

   vf = __fsqrt_rn(1.0f - x2) * (cn[1] + 1.5f * fx * cn[2] +
        a[0] * Q * (1.0f + Q)/(Q - fx *
        Q + 1.0f) + a[1] * Q3 + a[2] * Q5 * (f12-f2+fx) +
        cn[3] * (5.0f * x2 - 1.0f) * f2*.5f + a[3]*(1.0f  +
        f2*(4.0f * f2 - fx + x2 - 10.0f) + 5.0f * fx) * Q7);

    potentials[sensor_id] += C*(t0*vf + r0*vr)/R2;

    __syncthreads();
    float sum = potentials[sensor_id];
    sum = blockReduceSum(sum);
    if (threadIdx.x == 0) out[dipole_id] = sum ;
}
```

All operations are executed on single-precision variables and where possible, higher-performance NVIDIA CUDA fast mathematical functions were used, such as the `rsqrt` for the reciprocal square root or the intrinsic function `__fsqrt_rn` for square root.

4. Combining visualisation and computation – the stream architecture. Traditional EEG processing systems follow the batch processing style of computation; data is input from a file, a set of algorithms are executed on the data and the results are saved into output files. Intermediate data is stored in memory. These systems are typically CPU-centric, i.e. the CPU controls the entire execution flow as well as performs the heavy mathematical computations. This type of software architecture is the result of abstractions based on the common computer architecture model of the past few decades that are often commonly referred to as the "*Denial Architecture*" [30]. These systems pretend that computers operate sequentially and that memory architecture is flat and access is uniform. GPU architecture and the CUDA computing model explicitly expose the underlying parallel hardware components and encourages developers to take advantage of data locality and restructure instruction execution flow to discover new opportunities for increasing performance.

In addition, our GPU kernel implementations were used as co-processor functions with explicit data movement before and after the kernel calls. This resulted in a performance bottleneck, which was caused by executing visualisation calls and additional processing steps on the CPU.

Our software architecture is explicitly parallel. The underlying design is based the stream computing paradigm, in which data is processed in a parallel pipeline. Measurement data is pumped from the EEG measuremnt device or data file and passed through various loosely coupled processing nodes. The nodes are connected by FIFO channels and output data to further processing or visualization nodes. The processing nodes

can be executed by CPU threads or GPU kernels. The execution of the algorithms are triggered by the arrival of data and is carried out with maximum level of hardware parallelism. The pipeline operation and the coupling of CUDA kernels with the OpenGL graphics pipeline helped to remove any unnecessary data movement (by keep as much data as possible on the GPU).

Fig. 4.1 illustrates the high-level, abstract view of our architecture. The CPU system refers to the host computer that acts as a coordinator of the environment. Although it can contain several processing nodes, the goal is that most of the computation is carried out on the GPU and the host, ideally, only coordinates the execution process. The GPU system is responsible for computational and visualisation tasks. The output of this subsystem is the set of monitors comprising the display wall environment. Although not shown in the figure, but – if required – data can be fetched from the GPU and stored locally in the file system for future reference.

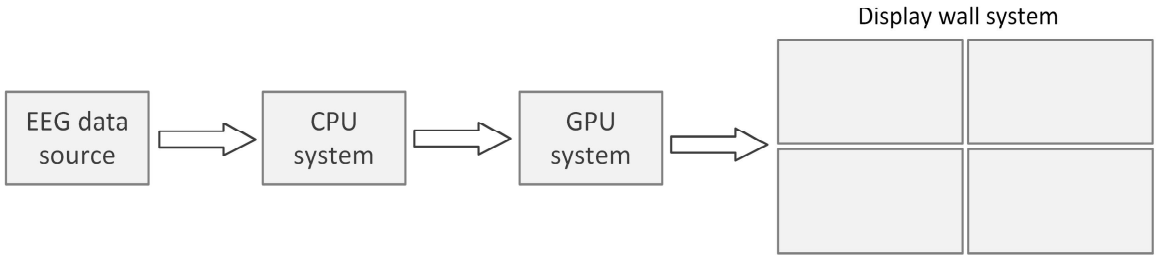


Fig. 4.1. *Simplified view of the CPU-GPU EEG processing and visualisation pipeline.*

The more detailed view of the architecture is depicted in Fig. 4.2. On the CPU side, green blocks mark objects that encapsulate the building blocks of the EEG processing system, e.g. head model, electrodes, computation, data source, etc. Although our system is implemented in Java, the principles can be carried over to implementations in C++ or other object-oriented languages. These objects allow the developer to configure the processing pipeline and parametrise the building blocks. The solid red arrow indicates the EEG stream data path, i.e., communication channels through which data travels from the source to the visualisation pipeline. The GPU system is accessed via interface layers; we use a JOGL wrapper layer for performing OpenGL calls and JCuda for calling CUDA functions from the Java objects.

We have avoided the use of any open-source visualization toolkit in order to retain maximum design freedom. Since the primary visualization aim is the display of time-varying potential or potential-derivative data on a static 2D or 3D geometry, head layer and electrode cap geometry data are cached into the GPU memory on initialization. Only potential data is updated continuously when the EEG marker position is changed. The OpenGL pipeline receives scalar values (potential or Laplacian results) as input and transforms these into vertex colour information via internal colour lookup tables implemented in the shader programs.

Potential data is processed and transformed by the GPU using CUDA kernels. The largest potential performance penalty in a GPU-based computational and visualization system is the host-GPU data transfer. If the GPU-computed results need to be copied back to the host in order to be displayed by a visualization system that will send the same data back to the GPU as part of the graphics calls, a major performance penalty is encountered. The CUDA-OpenGL interoperability architecture allows us to avoid this by directly modifying OpenGL data structures from inside the CUDA kernels.

Our algorithms are designed for scalability. As new GPU processors appear with more compute cores, the implementation automatically scales. If needed, the system can run with several GPU cards installed, in which case computation and visualization can be assigned to separate GPU devices.

5. Visualisation. The visualisation system is built around our display wall system consisting of four 46" Samsung edge-lit display panels arranged in a 2×2 grid. The resolution of the panels is 1920×1080 pixels, providing an overall 4K resolution drawing canvas for our application. The system is driven by an Intel Core-i5 hosted NVIDIA Quadro K4200 card (1344 cores, 2.1 teraflops). The surface area of the display wall – as shown in Fig. 5.1 – is sufficient to simultaneously display the EEG measurement plots, the 2D and 3D potential

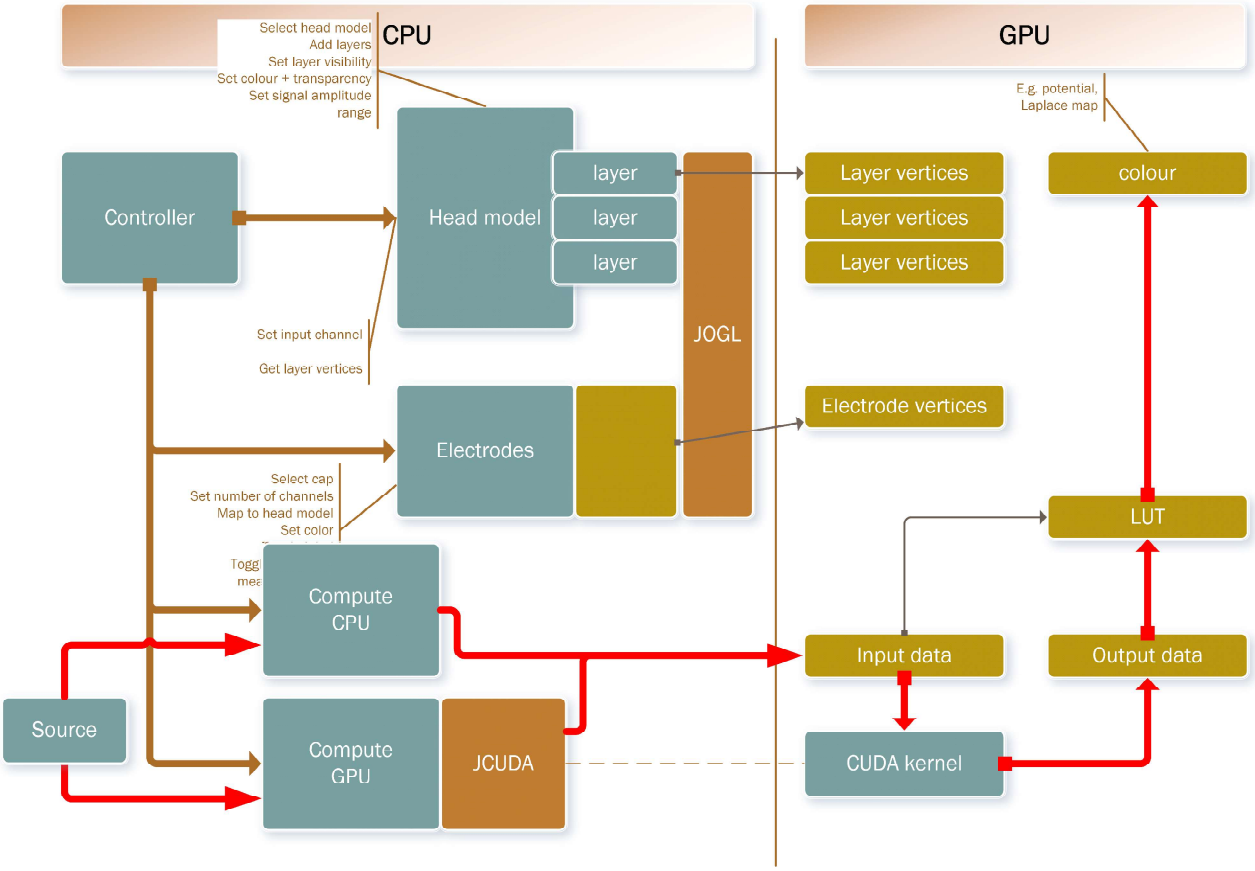


FIG. 4.2. *The high-level view of the GPU stream architecture. Solid arrows indicate data movement, thick red arrow depicts live data update path (stream). Data copied to the GPU memory are available to both the OpenGL and the CUDA backend. Processing node may or may not have GPU implementation parts. Their number and order are not limited.*

and Laplacian maps, as well as MRI images and (if available) video recordings. The visualization subsystem is implemented in OpenGL and JavaFX. OpenGL is used for all 3D operations, whereas JavaFX is for image views and general user interface elements.

The EEG plot (Fig. 5.2) displays the measured or pre-processed EEG data and allows the user to change the signal amplitude range and time axis resolution. The current sample to be processed is indicated by and selected with a marker (red vertical line). The marker can be moved to arbitrary positions to examine the activity map. It is also possible to play back the measurement and recompute the activity maps on the fly at the selected speed. Fig. 5.2 also illustrates the result of the surface Laplacian computation and its visualization as a 2D plot. The focusing effect the of the Laplacian is clearly visible.

5.1. Polygon mesh model. The core component of the visualisation subsystem is the polygon-mesh head model. Although the computational algorithms utilise the spherical head model, our display module is designed to support patient-specific realistic head models. Individual head models are created using the Freesurfer [31] and Brainstorm packages. Freesurfer is used to segment the patient's MR image into scalp and brain (white matter and cortical surface) layers. The Freesurfer output surfaces are then further processed with Brainstorm to generate the inner and outer skull surfaces.

The module allows one to select which head layers to display, adjust the transparency value of each layer if multiple layers are chosen. The results are fed into the OpenGL pipeline as a float value per vertex and converted into a vertex colour within the shader program. At the moment, the standard Jet (Fig. 5.3.a) and Blue-White-Red (Fig. 5.3.b) colour lookup tables are supported. It is also possible to choose on which layer

FIG. 5.1. *The EEG visualisation system in operation showing the EEG chart, surface Laplacian map, MRI slice and 3D volume rendering modalities.*

to visualise the result of the processing steps (potential or Laplacian maps), as shown in Figs. 5.3.a and 5.4.a. While at the moment cortical visualisation is the result of a simple projection of the scalp values, it creates the basis of more complex cortical imaging algorithms, such as distributed dipole model based inverse solutions, intracranial EEG visualisation and so on. The polygon view module also supports the usual rotation and zoom functions, hence any part of a surface can be examined in detail (Fig. 5.4.b).

5.2. MRI Viewer. In order to assist the users with anatomical data, the system also includes an MRI slice viewer module. The head can be viewed in axial, sagittal or coronal orientation, as well as in a combined view of these three orientations. The user can select how many slices to view (from 1 to 5×10 slices) – see Fig. 5.5 –, and can page or step through slices. If required, image contrast and intensity can be adjusted for the slices. The supported input formats are DICOM and Nifti. Images are input using the LONI Java Image IO Plugins [32] and displayed as JavaFX `ImageView` instances. Image slices are pre-processed (sorted, transformed) based on metadata in the MRI file.

5.3. 3D volumetric visualisation. On top of the slice viewer, the visualisation module also supports volumetric rendering of MRI brain images. A 3D texture-based ray-casting algorithm is implemented in OpenGL. Several options are incorporated for voxel visualisation, such as iso-surface or maximum intensity projection. A head model using the iso-surface approach is shown in Fig. 5.6. Note that the image was taken from a patient wearing the EEG electrode cap. The visibility of the cap provides the basis for a practical feature in our system. A common problem in EEG processing is the accurate registration of the EEG electrode coordinates on the patient's head. Normally these coordinates are estimated from the standard electrode coordinate map after measuring standard fiducial points (nasion, preauricular left and preauricular right) [33] or measured by expensive 3D digitising or scanner devices [34, 35].

In our system we implemented a 3D picking capability in the iso-surface volumetric rendering mode. As

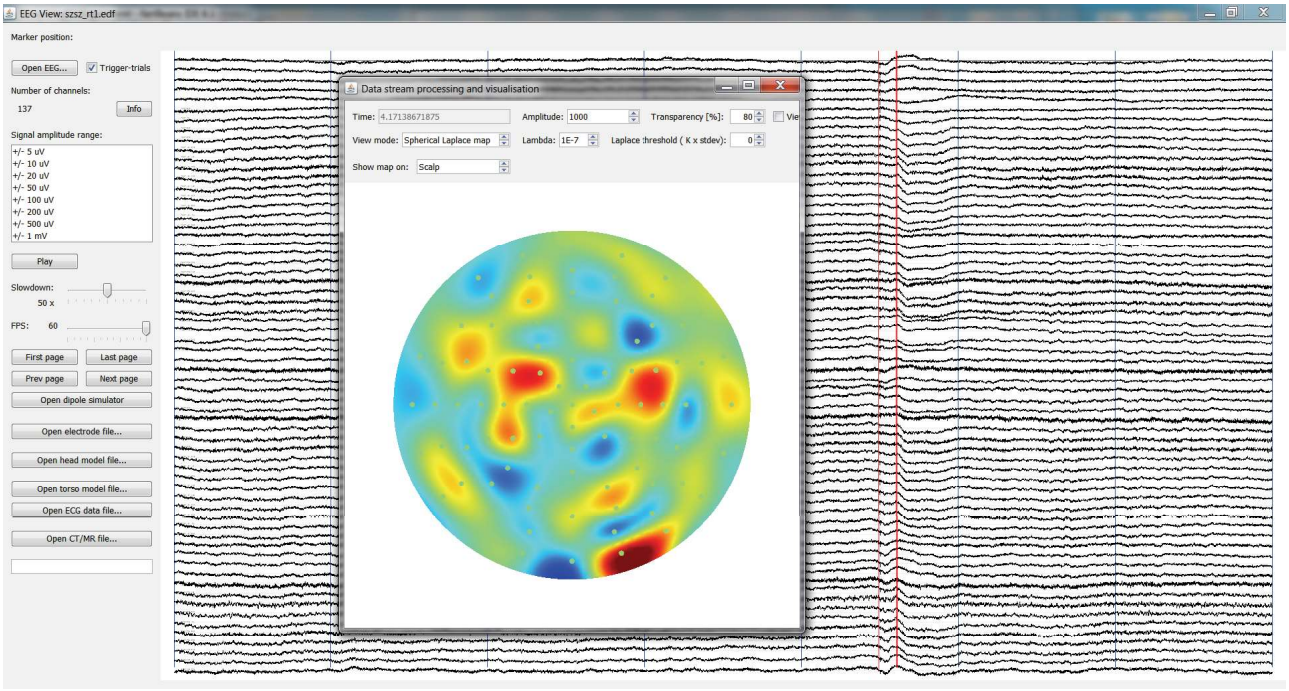
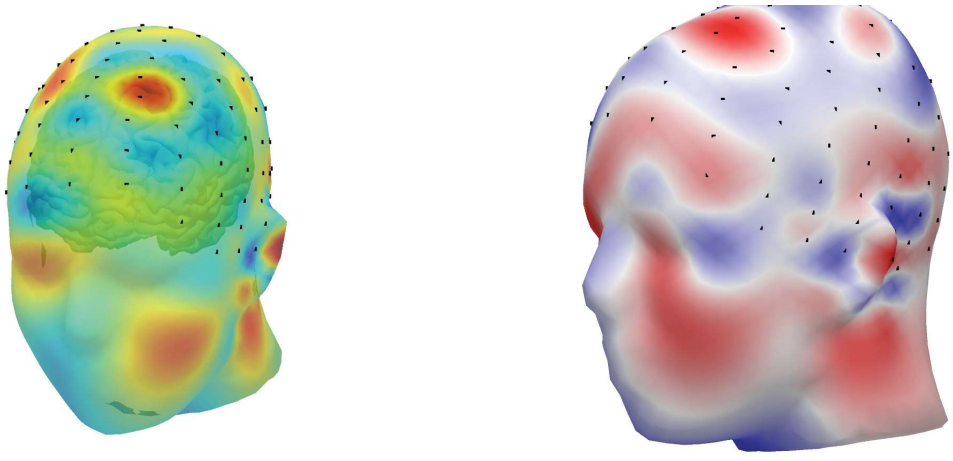


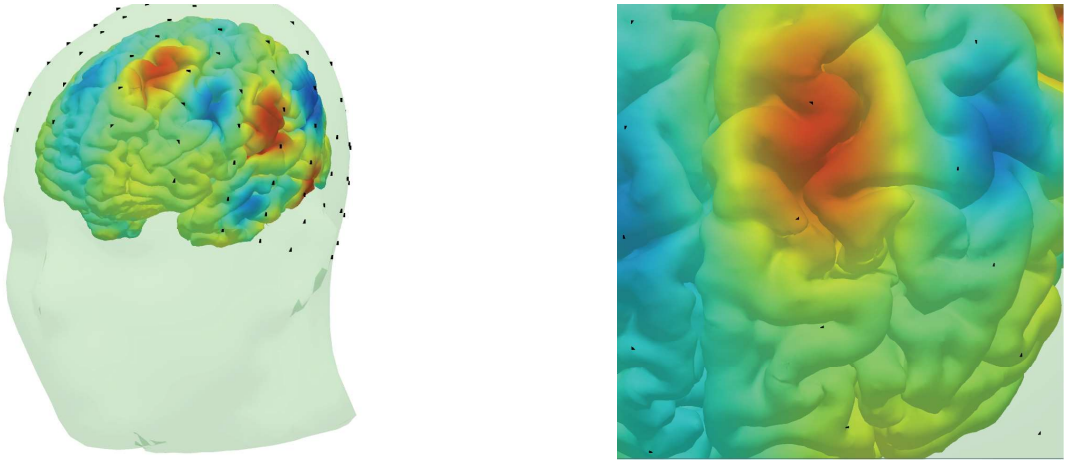
FIG. 5.2. *Screenshot of the prototype showing the multi-channel EEG plot and the 2D Laplacian map computed from the potentials.*



(a) (b)

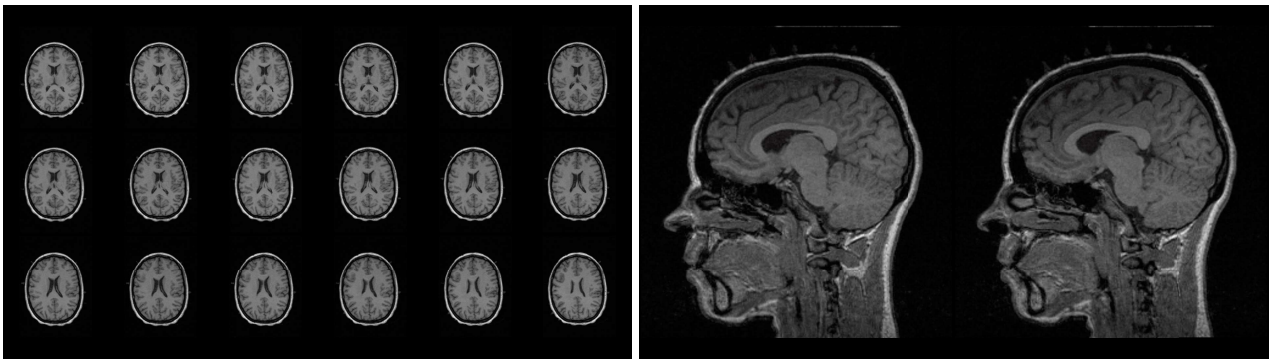
FIG. 5.3. *(a) Laplacian with Jet colour map: The surface Laplacian displayed on the scalp surface using the Jet colour lookup table. Red indicates positive, Blue negative values. (b) Laplacian with Red-Blue colour map: The surface Laplacian displayed on the scalp surface using the Blue-White-Red colour lookup table. Red indicates positive, Blue negative values.*

shown in Fig. 5.6, the centre of each electrode can be located on the 3D model. Each electrode centre is picked and the ray-surface intersection point is calculated and returned as the electrode coordinate. In order to achieve accurate scalp intersection calculation, for each electrode, the surface threshold is modified to remove the electrode 'cone' (Fig. 5.6). The processed electrode is marked with a small semi-transparent sphere to track progress. The system supports electrode coordinate labelling by using a user-selected electrode cap coordinate map that is used to locate the closest set of neighbour electrodes for the pick location.



(a) (b)

FIG. 5.4. *(a) The Laplacian map displayed on the cortex using back projection from the scalp; (b) Zoomed-in view of the cortical layer.*



(a) (b)

FIG. 5.5. *MRI slice viewer componen: (a) Axial slices displayed in a* 3×6 *arrangement; (b) Sagittal slices in a* 1×2 *layout arrangement*

FIG. 5.6. *3D volumetric reconstruction of the patient MRI slices with different surface threshold (left and middle) values, and progress in the electrode localisation 'pick' operation (right).*

TABLE 6.1
GPU-based surface Laplacian execution times (128 electrodes)

Number of surface points	Execution time [*msec*] n=15	n=50
1024	0.164	0.361
2048	0.166	0.379
4096	0.172	0.469
8192	0.189	1.033
16384	0.341	3.482
32768	0.677	5.941

TABLE 6.2
GPU-based spherical forward solution execution times

Number of dipoles	Execution time [*msec*] Cuffin & Cohen E=64	Cuffin & Cohen E=128	Sun-Stok E=64	Sun-Stok E=128
1	0.341	0.720	0.144	0.497
10	0.402	0.891	0.152	0.578
100	0.626	1.337	0.293	1.086
1000	1.721	3.374	0.444	1.761
10000	8.852	17.002	0.752	3.001

6. Results and conclusions. In addition to the novel, stream-based EEG processing software architecture and the wide range of visualisation capabilities presented in Sections 4 and 5, the most notable result of our development is the achieved computational performance for the selected two key EEG processing algorithms. Our system has achieved considerable increase in computational performance. Tables 6.1 and 6.2 show the achieved execution times with the parallel GPU Laplace and forward implementations. Results were obtained using an NVIDIA Quadro K4200 GPU card having 1344 cores.

As shown in Table 6.1, using 128 electrodes and up to 2048 ($N = 50$) or 8192 ($N = 15$) surface points, the system can compute and visualize the Laplacian map in real-time at up to 2 kHz sampling frequency. Note that execution times are given in milliseconds, not in seconds! When further increasing the number of surface points, the system will work in soft real-time mode, still simultaneously computing and visualising results but at a lower rate.

The performance results for the forward problem are similarly impressive, as shown in Table 6.2. Using several code optimization steps we were able to improve the performance of the GPU-version of the Cuffin-Cohen algorithm [28]. Using Suns algorithm we could further decrease the execution time and reach millisecond range for 128 channels and 5000 dipoles.

We described a novel, GPU-based stream-oriented EEG processing software system. We demonstrated that the GPU architecture is well suited to simultaneous EEG visualization and processing. The computational performance offered by current GPUs make it possible to carry out simultaneous computation and visualization in real-time, or online if live measurement is performed. We were also interested in exploring new architectural and performance optimization methods that could lead towards the development of EEG software systems with higher flexibility and several orders of magnitude larger performance than what is available today. The removal of unnecessary host-GPU data transfers, optimised memory access and instruction execution strategies all contributed to the high achieved computational performance.

We have illustrated using two important EEG imaging algorithms, how a GPU-based stream-oriented system can be created, and showed that computational performance up to three orders of magnitude higher than MATLAB can be achieved while providing simultaneous interactive 2D/3D visualization. For both algorithms, the single-precision GPU implementation produced results with $RMSE < 2.6 \times 10^{-3}$ when compared to the MATLAB implementations.

The high-performance simultaneous processing and visualisation system integrated with a 4K resolution 2×2 display wall environment creates a very fast, efficient, yet cost-effective brain imaging system. Future development plans include the integration and coupling of other imaging modalities such as MRI and fMRI

in our framework, and implementing fast realistic head-model based cortical imaging algorithms. Our fast implementation of the surface Laplacian can also serve as a basis for high-resolution cognitive studies and BCI applications. The software architecture and the system can be easily adapted for use in ECG-based body-surface potential mapping applications or similar computationally intensive visualisation areas.

Acknowledgements. The authors wish to thank Prof Zoltan Nagy and the National Institute for Clinical Neurosciences (Budapest, Hungary) for providing access to patient MRI and EEG measurement data, as well as their students, Andras Gergely Nagy (MSc) and Daniel Kenyeres (BSc) for their invaluable help in the development of the system described in this paper.

REFERENCES

[1] C. M. Michel, M. M. Murray, *Towards the utilization of EEG as a brain imaging tool*, NeuroImage, 61 (2012), pp. 371-385.
[2] L. Ding, G. A. Worrell, T. D. Lagerlund, and B. He, *3D source localization of interictal spikes in epilepsy patients with MRI lesions.*, Phys. Med. Biol., vol. 51, no. 16, pp. 4047-62, Aug. 2006.
[3] B. Alkonyi, C. Juhász, O. Muzik, E. Asano, A. Saporta, A. Shah, and H. T. Chugani, *Quantitative brain surface mapping of an electrophysiologic/metabolic mismatch in human neocortical epilepsy.*, Epilepsy Res., vol. 87, no. 1, pp. 77-87, Nov. 2009.
[4] N. Srinivasan, *Cognitive neuroscience of creativity: EEG based approaches*, Methods, vol. 42, no. 1, pp. 109-116, May 2007.
[5] Z. A. Acar and S. Makeig, *Neuroelectromagnetic forward head modeling toolbox.*, J. Neurosci. Methods, vol. 190, no. 2, pp. 258-70, Jul. 2010.
[6] R. Oostenveld, P. Fries, E. Maris, and J. Schoffelen, *FieldTrip: Open Source Software for Advanced Analysis of MEG, EEG, and Invasive Electrophysiological Data*, Computational Intelligence and Neuroscience, vol. 2011, Article ID 156869, 9 pages, 2011. doi:10.1155/2011/156869
[7] J. Kayser and C. E. Tenke, *Principal components analysis of Laplacian waveforms as a generic method for identifying ERP generator patterns: I. Evaluation with auditory oddball tasks.*, Clin. Neurophysiol., vol. 117, no. 2, pp. 348-368, Feb. 2006.
[8] F. Perrin, J. Pernier, O. Bertrand, and J. F. Echallier, *Spherical splines for scalp potential and current density mapping*, Electroencephalogr. Clin. Neurophysiol., vol. 72, no. 2, pp. 184-187, 1989.
[9] N. Cuffin and D. Cohen, *Comparison of the Magnetoenchephalogram and Electroencephalogram*, Electroencephalogr. Clin. Neurophysiol., vol. 47, pp. 132-146, 1979.
[10] R. Ramalho, P. Tomás, and L. Sousa, *Efficient Independent Component Analysis on a GPU*, in 2010 10th IEEE International Conference on Computer and Information Technology, 2010, pp. 1128-1133.
[11] M. Corraine, S. Lopez, and L. Wang, *GPU Acceleration of Transmural Electrophysiological Imaging*, pp. 849-852, 2012.
[12] C. Dinh, J. Ruhle, S. Bollmann, and D. Gullmar, *A GPU-accelerated Performance Optimized RAP-MUSIC Algorithm for Online Source Localization.*, Biomedizinische Technik (Berl.) (2012), 57
[13] T. De Marco, F. Ries, M. Guermandi, and R. Guerrieri, *EIT forward problem parallel simulation environment with anisotropic tissue and realistic electrode models.*, IEEE Trans. Biomed. Eng., vol. 59, no. 5, pp. 1229-39, May 2012.
[14] R. Trobec, *Computer analysis of multichannel ECG*, Comput. Biol. Med., vol. 33, no. 3, pp. 215-226, May 2003.
[15] B. He, G. Li, and J. Lian, *A spline Laplacian ECG estimator in a realistic geometry volume conductor.*, IEEE Trans. Biomed. Eng., vol. 49, no. 2, pp. 110-7, Feb. 2002.
[16] G. Li, J. Lian, and B. He, *Spatial resolution of body surface potential and Laplacian pace mapping.*, Pacing Clin. Electrophysiol., vol. 25:(4 Pt 1), pp. 420-9, Apr. 2002.
[17] P. L. Nunez and R. Srinivasan, *Electric Fields of the Brain: The Neurophysics of EEG*, 2nd Edition. Oxford University Press, USA, 2005.
[18] P. L. Nunez and K. L. Pilgreen, *The spline-Laplacian in clinical neurophysiology: a method to improve EEG spatial resolution.*, J. Clin. Neurophysiol., vol. 8, no. 4, pp. 397-413, Oct. 1991.
[19] P. L. Nunez and A. F. Westdorp, *The surface laplacian, high resolution EEG and controversies*, Brain Topogr., vol. 6, no. 3, pp. 221-226, Mar. 1994.
[20] *Corrigenda*, Electroencephalogr. Clin. Neurophysiol., vol. 76, no. 6, pp. 565-566, Dec. 1990.
[21] R. Grech, T. Cassar, J. Muscat, K. P. Camilleri, S. G. Fabri, M. Zervakis, P. Xanthopoulos, V. Sakkalis, and B. Vanrumste, *Review on solving the inverse problem in EEG source analysis.*, J. Neuroeng. Rehabil., vol. 5, p. 25, Jan. 2008.
[22] C. M. Michel, M. M. Murray, G. Lantz, S. Gonzalez, L. Spinelli, and R. Grave de Peralta, *EEG source imaging.*, Clin. Neurophysiol., vol. 115, no. 10, pp. 2195-222, Oct. 2004.
[23] H. Hallez, B. Vanrumste, R. Grech, J. Muscat, W. De Clercq, A. Vergult, Y. DAsseler, K. P. Camilleri, S. G. Fabri, S. Van Huffel, and I. Lemahieu, *Review on solving the forward problem in EEG source analysis.*, J. Neuroeng. Rehabil., vol. 4, p. 46, Jan. 2007.
[24] O. Hauk, *Keep it simple: a case for using classical minimum norm estimation in the analysis of EEG and MEG data.*, Neuroimage, vol. 21, no. 4, pp. 1612-21, Apr. 2004.
[25] P. Berg and M. Scherg, *A fast method for forward computation of multiple-shell spherical head models*, Electroencephalogr. Clin. Neurophysiol., vol. 90, no. 1, pp. 58-64, Jan. 1994.

[26] *OpenACC Home — www.openacc.org* [Online]. Available: http://www.openacc.org/. [Accessed: 20-Dec-2015].
[27] NVIDIA Corp., *Cuda C Programming Guide*, Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide. [Accessed: 20-Dec-2015]
[28] N. B. Bangera and D. Lewine, *Accelerated large scale spherical model forward solutions for the EEG / MEG using CUDA*, in GPU Technology Conference, 2010, vol. 23, no. 11, p. Poster.
[29] M. Sun, *An efficient algorithm for computing multishell spherical volume conductor models in EEG dipole source localization.*, IEEE Trans. Biomed. Eng., vol. 44, no. 12, pp. 1243-52, Dec. 1997.
[30] B. Dally, *The End of Denial Architecture and The Rise of Throughput Computing*, Electronics, 2009.
[31] B. Fischl, *FreeSurfer*, Neuroimage, vol. 62, no. 2, pp. 774-81, Aug. 2012.
[32] *LONI Java Image I/O Plugins — Laboratory of Neuro Imaging.* [Online]. Available: http://www.loni.usc.edu/Software/IO_Plugins. [Accessed: 20-Dec-2015].
[33] U. Windhorst and H. Johansson, *Modern Techniques in Neuroscience Research*. Springer Science & Business Media, 2012.
[34] *xensorTM — ANT Neuro.* [Online]. Available: https://www.ant-neuro.com/products/xensor. [Accessed: 20-Dec-2015].
[35] S. S. Dalal, S. Rampp, F. Willomitzer, and S. Ettl, *Consequences of EEG electrode position error on ultimate beamformer source reconstruction performance*, Front. Neurosci., vol. 8, p. 42, Jan. 2014.
[36] M. J. Kilgard, *Modern OpenGL Usage: Using Vertex Buffer Objects Well*, White paper, Available: https://www.researchgate.net/publication/240320227_Modern_OpenGL_usage_using_vertex_buffer_objects_well

Edited by: Karolj Skala
Received: December 21, 2015
Accepted: March 31, 2016