GPU-accelerated stress analysis in geomechanics

Attila M. Zsaki
Department of Building, Civil and Environmental Engineering – Concordia University, Montreal, Quebec, Canada

ABSTRACT
Design in geomechanics is often data-limited. This is alleviated by many simulation runs to account for the variability of material properties or the modeling of an extensively sized domain to evaluate the response of soils and rocks. Both approaches require considerable computational resources and time. The current line of graphics processing units is offering an alternative to traditional parallel computers. Through an illustrative example, this paper formulates the computation of field quantities using the boundary element method to run on graphics processing units. As a result, the computation of stresses and displacements around underground excavations were sped up by an order of magnitude.

PRESENTACIONES TÉCNICAS
Los diseños geomecánicos están frecuentemente muy limitados en lo referente a sus parámetros de entrada. Esto es compensado, ya sea con la simulación de muchos escenarios para lograr tener en cuenta la variabilidad en las propiedades de los materiales, o con la modelación de dominios de gran dimensión para evaluar la respuesta de suelos y de rocas. Ambas opciones requieren recursos computacionales y tiempo considerables. Las unidades de procesamiento gráfico disponibles en el momento ofrecen una alternativa a la tradicional computación paralela. A través de un ejemplo ilustrativo, esta publicación formula el uso de unidades de procesamiento gráfico para el cálculo de modelos empleando el método de elementos de frontera. Como resultado, el tiempo de cálculo de esfuerzos y de desplazamientos en las zonas cercanas a excavaciones subterráneas fue reducido en un orden de magnitud.

1 INTRODUCTION

Perhaps what sets apart modeling in geomechanics from other engineering disciplines is the general lack of available data to characterize geomaterials throughout the domain modeled, and the extent and material boundaries within the domain, particularly for three-dimensional analyses. These difficulties are often addressed by using probabilistic simulations to sample material properties over their credible range and compute stresses and displacements throughout a region in the soil or rock. Common to these approaches is that both require significant computational effort. The sampling process, even for a few key material properties requires a large number of samples from a distribution to arrive at statistically satisfactory representation of the outcome. While in a boundary element formulation, the response of a rock mass is evaluated by computing field quantities at a large number of points.

Fortunately both of these scenarios are such that each sampling and in the probabilistic simulation or calculation of stresses at a field point can be done independently of any other sample or point in the domain. Traditionally this was exploited to draft algorithms that can be cast to run in parallel and thus these were aptly named as 'embarrassingly parallel' for this reason (Zsaki and Curran 2002). However, even at this time the cost and availability of computational resources can hinder the widespread use of such approaches.

In the recent years, however, the emergence of powerful commodity graphics hardware with many processing units (GPUs) potentially enables the same computations to be performed at the fraction of the cost but with significant speedups. This paper aims at highlighting the capability of GPUs for use in calculations often performed in geomechanics. An illustrative example demonstrates the speedup achievable and the implementation issues.

2 PARALLEL COMPUTATION ON GPUS

The GPUs, found in computers today, were designed to meet the growing needs of real-time 3D graphics and gaming industry. Internally, these devices are multi-cored, multi-threaded, highly parallel processors (NVIDIA Corp. 2008). Their massive parallelism is suitable to graphics processing since many operations can be executed independently and the same operation is performed on many pixels or vertices at the same time giving rise to potentially high parallelism. In parallel processing terminology, their operation is fine-grained data-parallel and thread-parallel (NVIDIA Corp. 2008). These devices communicate with the host CPU over a high-bandwidth memory interconnect. Internally, the GPU's operation is characterized by a thread that carries out the instructions. Threads are the basic building blocks with access to fast register memory and a considerably slower global memory of the GPU (NVIDIA Corp. 2008). A group or block of threads has access to a shared memory, which can be used to foster cooperation between threads. Communication between the GPU (often called device) and the CPU (referred to as host) is through a global memory. The practical implementation of a programming model requires allocation and transfer of memory storage for variables from the host to device, execution of the

computation in the GPU using a kernel, and finally the memory transfer of results back to the host. Development using the NVIDIA compiler enables coexistence of C and GPU code in the same source file. The definition of GPU specific code is done using the CUDA (Compute Unified Device Architecture) directives and instructions (NVIDIA Corp. 2008). Thus separation of GPU code is accomplished using classifiers such as __global__ or __device__. CUDA contains most of the common mathematical functions available from C with the exception of a few such as a random number generator, for which a third party code can be used. Precision in floating point computation can be achieved using single precision or on more recent GPU architectures with double precision, which conforms to the IEEE-745 standard (NVIDIA Corp. 2008). GPUs can run a large number of threads in parallel. However, there are limitations on the number of threads in a block, which is 512 for the current generation of GPUs (NVIDIA Corp. 2008). All these might change with the next generation of GPU architecture. The choice of the number of threads can be critical; if too few are used, the available parallel performance is not exploited, while if too many are scheduled, competition for resources can ensue. Also, it has to be kept in mind that the resources of a GPU are not allowed to be tied up for long stretches of time since it needs to refresh and redraw the screen as well. Commonly, the time limit is set at 5 sec (NVIDIA Corp. 2008). However, the GPU can schedule thread execution interleaved with other duties. Therefore the execution of a single thread should be less than this time limit. An alternative is to use graphics processing cards based on NVIDIA's TESLA architecture, which are compute only, effectively lifting the time limitation (NVIDIA Corp. 2008). Development using GPUs is not that dissimilar from other parallel programming development. However, it contains enough intricacies to warrant the demonstration of its basic principles using an example problem drawn from geomechanics.

3 COMPUTATION OF FIELD QUANTITIES IN A BOUNDARY ELEMENT CODE

The practical example chosen addresses a class of common computation in geomechanics, where the response of a geologic medium is investigated in the presence of underground excavations. Commonly this class of problems is modeled using the finite (FEM) or boundary element method (BEM). For a BEM approach, the solution is computed on the surface of an excavation and/or free boundary, however the main interest lies in the displacements and stresses (field quantities) within the medium. Often this response is captured at discrete points, be that in 2D or 3D, or field points. Without being specific about the formulation of the BEM or the dimensionality of problem, a computation of field quantities at any given point can be performed independently of calculation at any other point's response. This could lend itself to a massively parallel implementation. As an example, a 2D BEM for elastic problems using constant elements, as appears in Kythe (1995), is adopted for GPU realization. Although this formulation is quite simple and might not be applicable to advanced modeling, it contains the essence of the BEM and since the goal of this paper is to present a GPU implementation rather than advocate the suitability of any particular method. If desired, other types of BEM formulations, once the discrete form of equations is derived, can easily implemented on GPUs.

Given a domain and its boundary discretized into constant elements, such as shown on Figure 1, the essence of the BEM can be expressed using the notation developed in Kythe (1995).

Equation 1 expresses the equilibrium of contributions from displacements (u), body forces (B) and tractions (p). The detailed discussion of the derivation of equation 6 is found in Kythe (1995).

$$\sum_{j=1}^{N} H_{ij} u_j = B_i + \sum_{j=1}^{N} G_{ij} p_j \qquad [1]$$

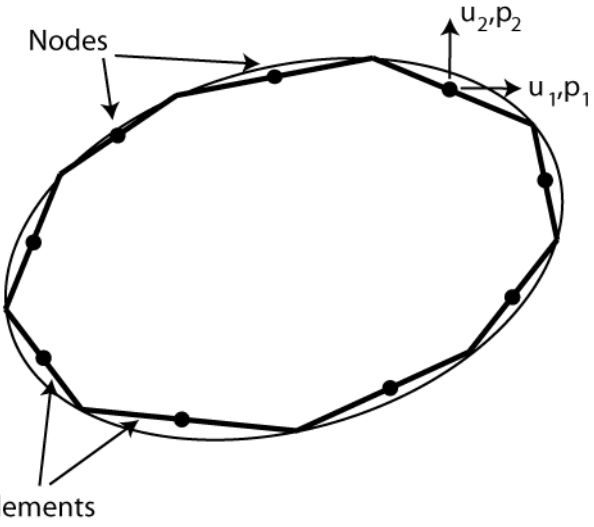

Figure 1. Domain discretized with constant boundary elements, after Kythe (1995)

Once the solution of Equation 1 is computed, response of the system at field points can be obtained using Equations 2 and 3 for displacements and stresses, respectively, with additional quantities defined in equations 4 through 6. Quantities used in these equations are those derived and defined in Kythe (1995), and their detailed discussion in this paper is not presented since any BEM formulation can be used.

$$u(i) = \sum_{s=1}^{M} \sum_{k=1}^{l} (u\ b)_k w_k\ A_s + \sum_{j=1}^{N} \left\{ \int_{C_j} u\ ds \right\} p_j - \sum_{j=1}^{N} \left\{ \int_{C_j} p\ ds \right\} u_j \quad [2]$$

$$_{ij} = \int_R D_{ij}\, b\, dx_1 dx_2 + \int_C D_{ij} p\, ds - \int_C S_{ij}\, u\, ds = \sum_{s=1}^{M} \left\{ \int_R D_{ij}\, dx_1\, dx_2 \right\} b_s + \sum_{j=1}^{N} \left\{ \int_{C_j} D_{ij}\, ds \right\} p_j - \sum_{j=1}^{N} \left\{ \int_{C_j} S_{ij}\, ds \right\} u_j \quad [3]$$

where, in two dimensions

$$D_{ij} = [D_1\, D_2]\ S_{ij} = [S_1\, S_2] \quad [4]$$

$$p = [p_1\, p_2]^T\ u = [u_1\, u_2]^T$$

and for k=1,2

$$D_k = \frac{1}{4\ (1\quad)r\ +2r_{,i}r_{,j}r_{,k}} (1\quad 2\ )\{\ _{ki}r_{,j} + \ _{kj}r_{,i} \quad _{ij}r_{,k}\} \quad [5]$$

$$S_k = \frac{}{2\ (1\quad)r^2} \begin{array}{l} (1\quad 2\ )\ _{ij}r_{,k} \\ 2\dfrac{r}{n}\ +\ (\ _{ik}r_{,j} + \ _{jk}r_{,i}) \ + \\ 4r_{,i}r_{,j}r_{,k} \\ 2\ \left(n_i r_{,j} r_{,k} + n_j r_{,i} r_{,k}\right) \\ +(1\quad 2\ ) \begin{array}{l} 2n_k r_{,i} r_{,j} \\ +n_j\ _{ik} + n_i\ _{jk} \end{array} \div \\ (1\quad 4\ )n_k\ _{ij} \end{array} \quad [6]$$

The previous set of equations, given the solution at boundaries, is evaluated at every field point. In a serial CPU code, a loop is created to compute stresses and displacements for the field points. The BEM code is a modified version of the one found in Kythe (1995), which should be consulted for exact definition and meaning of variables. The salient features of this implementation are summarized in Table 1.

Table 1. CPU serial code for BEM field point computations

```
// Compute stress and displacement at field points

//  For all field points
for (k=1;k<=L;k++) {

  // For all boundary elements
  for (j=1;j<=N;j++) {

    kk=j+1;

    // Gaussian quadrature to compute the elements
    //  of H and G
    Quad11(Xi[k],Yi[k],X[j],Y[j],X[kk],Y[kk],&H11,&H12,
           &H21,&H22, &G11,&G12,&G22);

    // Computing the displacement values
    displ[2*k-1]+=F[2*j-1]*G11+F[2*j]*G12-Bc[2*j-1]*H11
                  -Bc[2*j]*H12;
    displ[2*k]+=F[2*j-1]*G12+F[2*j]*G22-Bc[2*j-1]*H21
                -Bc[2*j]*H22;

    // Computing the coefficients used in
    //  stress computation
    Stress(Xi[k],Yi[k],X[j],Y[j],X[kk],Y[kk],&dx11,
           &dy11,&dx12,&dy12,&dx22,&dy22,&sx11,
           &sy11,&sx12,&sy12,&sx22,&sy22);

    // Computing the stress values
    stress[3*k-2]+=F[2*j-1]*dx11+F[2*j]*dy11-Bc[2*j-
                   1]*sx11-Bc[2*j]*sy11;
    stress[3*k-1]+=F[2*j-1]*dx12+F[2*j]*dy12-Bc[2*j-
                   1]*sx12-Bc[2*j]*sy12;
    stress[3*k]+=F[2*j-1]*dx22+F[2*j]*dy22-Bc[2*j-
1]*sx22
                  -Bc[2*j]*sy22;
  }
}
```

The preceding code fragment encapsulates all the major function calls and statements describing the computation of field quantities. The BEM code was run fifty times to gather performance statistics on a problem shown on Figure 2, which models a circular hole embedded in an infinite elastic medium. The location of field points is shown in the shaded area. The tests used one million field points equally distributed in the enclosed area. Average computation time was 34.73 sec with a standard deviation of 0.016 sec on a current generation MacBook Pro.

The strategy to convert the code for GPU is as follows; the computation is moved from the CPU to the GPU with the additional expenses of memory allocation on the GPU and memory transfer to and from the GPU. The pseudo code for GPU implementation is shown in Table 2

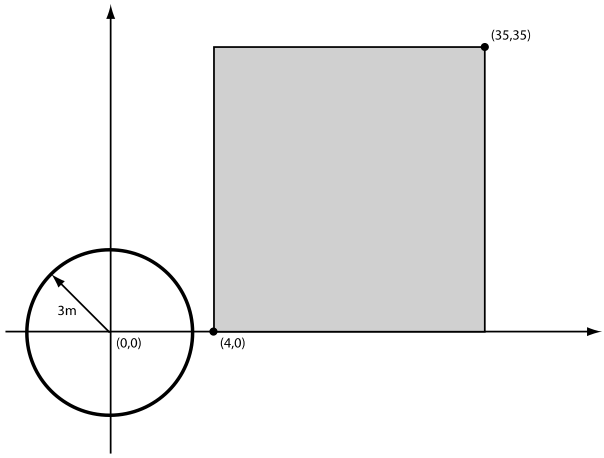

Figure 2. A circular hole in an infinite medium and location of field points for computation

Table 2. GPU code for field quantity computations

```
// Main GPU kernel
__global__ void Inter11GPUKernel(float* Bc, float* F,
float* Xi, float* Yi, float* X, float* Y, float* stress, float*
displ, int N, int L, float nu, float mu)
{
 // Determining thread ID
 idx=blockIdx.x*blockDim.x+threadIdx.x;

  idxPlus1=idx+1;

  if (idxPlus1<L+1) {

    // For all boundary elements
    for (j=1;j<=N;j++) {

      kk=j+1;

      Quad11Kernel(Xi[idxPlus1],Yi[idxPlus1],
              X[j],Y[j],X[kk],Y[kk],&H11,
              &H12,&H21,&H22,&G11,&G12,
              &G22,nu,mu);

      displ[2*idxPlus1-1]+=F[2*j-1]*G11+F[2*j]*G12
                           -Bc[2*j-1]*H11-Bc[2*j]*H12;
      displ[2*idxPlus1]+=F[2*j-1]*G12+F[2*j]*G22
                         -Bc[2*j-1]*H21-Bc[2*j]*H22;

     StressKernel(Xi[idxPlus1],Yi[idxPlus1],X[j],Y[j],X[kk],
       Y[kk],&dx11,&dy11,&dx12,&dy12,&dx22,
       &dy22,&sx11,&sy11,&sx12,&sy12,&sx22,
       &sy22,nu,mu);

      stress[3*idxPlus1-2]+=F[2*j-1]*dx11
                  +F[2*j]*dy11-Bc[2*j-1]*sx11-Bc[2*j]*sy11;
      stress[3*idxPlus1-1]+=F[2*j-1]*dx12+F[2*j]*dy12
                  -Bc[2*j-1]*sx12-Bc[2*j]*sy12;
      stress[3*idxPlus1]+=F[2*j-1]*dx22+F[2*j]*dy22
                  -Bc[2*j-1]*sx22-Bc[2*j]*sy22;
    }
  }
}

void main (…)
{
   …

  // Allocating arrays on the GPU
  size_t sizeForDevice_displ;
  sizeForDevice_displ=2*(L+1)*sizeof(float);
  cudaMalloc((void**)&displDevice,
                    sizeForDevice_displ);

  // Similarly allocating memory for: stressDevice,
  // XiDevice, YiDevice, XDevice, YDevice,
  // FDevice, BcDevice
  …

  // Transferring the contents of arrays to the GPU
  cudaMemcpy(displDevice,displ,
      sizeForDevice_displ,cudaMemcpyHostToDevice);
  cudaMemcpy(stressDevice,stress,
     sizeForDevice_stress,cudaMemcpyHostToDevice);
  …
  cudaMemcpy(BcDevice,Bc,sizeForDevice_Bc,
                              cudaMemcpyHostToDevice);

  // Establishing block size and number of threads
  numThreadsPerBlock=100;
  numBlocksPerGrid=L/numThreadsPerBlock
                  +(L%numThreadsPerBlock==0 ? 0:1);

  // Executing the main kernel on the GPU
  Inter11GPUKernel<<<numBlocksPerGrid,
        numThreadsPerBlock>>>(BcDevice,
        FDevice,XiDevice,YiDevice,
        XDevice,YDevice,stressDevice,
        displDevice,N,L,nu,mu);

  // Retrieving the results from the GPU
  cudaMemcpy(displ,displDevice,
      sizeForDevice_displ,cudaMemcpyDeviceToHost);
  cudaMemcpy(stress,stressDevice,
     sizeForDevice_stress,cudaMemcpyDeviceToHost);

  // Freeing the memory
  cudaFree(displDevice);
  cudaFree(stressDevice);
  …
  cudaFree(BcDevice);
```

Performance statistics gathered from fifty runs of the GPU code resulted in an average running time of 2.496

sec with a standard deviation of 0.153 sec on a MacBook Pro with a NVIDIA GeForce 8600M GT. The speedup achieved on the MacBook Pro was almost 14. The split of computation versus memory transfer for this case study was as follows; 2.5% of time for memory transfer and 97.5% for computation.

An important observation can be made from the statistics gathered; the amount of time devoted to memory transfer and to subsequent computation can significantly affect the relative speedup achieved in a GPU implementation. Ideally, the memory transfer should be kept at minimum. However, in practice this greatly depends on the problem being solved. Similarly, the number of field points to compute can affect the speedup achieved, as shown in Figure 3, in which for the BEM setup, the number of field points was varied from 1000 to 1000000 and the achieved speedup is plotted up.

The speedup realized for the BEM calculation starts with about 8 and rises to 14 for the one million field points. However, the early gains level off at about 100,000 points and remains relatively flat giving rise to the maximum estimate of speedup of about 14 for this type of computation on this hardware. Thus, as the number of field points increases the achieved speedup increases as well, with even the smallest sample set generates a speedup of almost 8. This can be attributed to the ratio of computation and memory transfer. Thus it can be concluded that the expected and achieved speedup is intimately dependent on the type of computation and its algorithmic realization. This gives an a priori estimate of how much speedup can be anticipated in a massively parallel implementation of an algorithm running on a GPU.

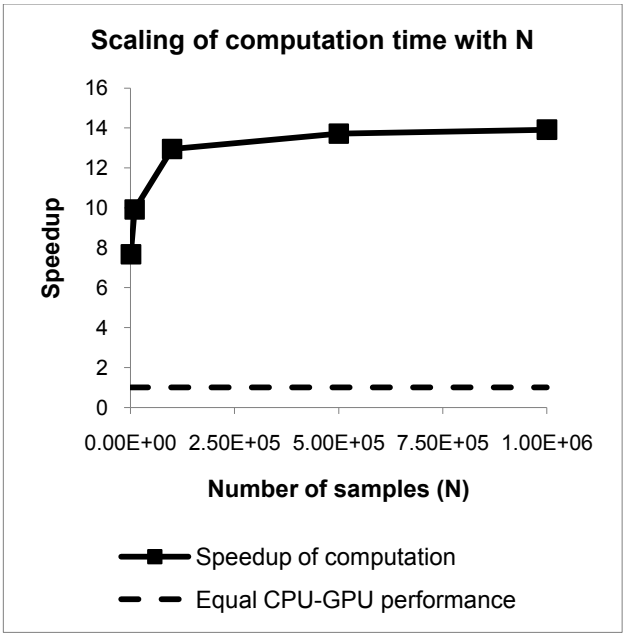

Figure 3. Speedup realized as a function of sample size in BEM field quantity computation

4 CONCLUSION

Computation in geomechanics is characterized by determining the behaviour of a system consisting of rocks and soils using some form of numerical method. Due to the inherent spatial extent and variability of properties of the constituent materials, the computation considers either repeated sampling of these properties for some class of problems or computing the response of a system at a large number of locations. Both cases require considerable computational effort. Traditional parallel computing with many CPUs was able to expedite computations, however the cost of large parallel computers could hinder widespread use of such systems. With the rapid advances in graphics hardware a new alternative is available; graphics processing units that can be used as a massively parallel computer. However, only certain types of computations can be efficiently adapted to run on GPUs. Primarily, where the inter-thread communication and memory transfer can be kept at the minimum. This paper introduced the current concepts of GPU computing using CUDA and presented an illustrative exampleof numerical stress analysis using a BEM formulation implemented on GPUs. The GPU example was tested against the CPU only implementation and the achieved speedup was above ten-fold. This represents a significant speedup considering that the required hardware is already part of the system potentially paving the way for more complex and detailed numerical modeling in geomechanics.

ACKNOWLEDGEMENTS

The research was supported by an NSERC Discovery grant held by the author.

REFERENCES

Kythe, P.K. 1995 *An introduction to boundary element methods*. CRC Press, Boca Raton, FL, USA.

NVIDIA Corp. 2008 *CUDA – Compute unified device architecture, Programming Guide*, Version 2.0. NVIDIA Corp., Santa Clara, CA, USA.

Zsaki, A.M. and Curran, J.H. 2002 Parallel computation of field quantities in an underground excavations analysis code. *5th NARMS and 17th TAC Conference*, Toronto, Ontario, Canada, 671-677.