Computing Boundary Element Method's Matrices on GPU

Samar Vafai[1], Martin Schanz[2] and Gundolf Haase[3]

[1] Institute of Mathematics and Scientific Computing, University of Graz
[2] Institute of Applied Mechanics, Graz University of Technology
[3] Institute of Mathematics and Scientific Computing, University of Graz**

Abstract. Matrices resulting from the boundary element method are dense and computationally expensive. To speed up the computational time, the matrix computation is done on GPU. The parallel processing capability of the Graphics Processing Unit (GPU) allows us to divide complex computing tasks into several thousands of smaller tasks that can be run concurrently. We achieved an acceleration of $23-43$ in comparison to a computation performed on the CPU, serially.

1 Introduction

Wave propagation is an important topic in engineering sciences, especially, in the field of solid mechanics. Applications of wave phenomena can be found in nearly in every field of engineering. In non-destructive testing, disturbances of travelling waves are measured to identify cracks or inclusions in the material. In the field of mining, blasting introduces intense stress waves to burst rocks or parts of it. Seismic waves are used to study the interior construction of the earth. Waves produced by earthquakes can cause tremendous destruction in buildings or other man made constructions. Therefor, knowledge is necessary how waves propagate in soil to prevent buildings or dams from destruction [1].

This short, certainly incomplete listing shows the importance of wave propagation problems in engineering mechanics. To tackle such problems correctly will lead to an improvement of constructions and higher quality of living by protecting houses from tremors [1].

In this study, the wave propagation in the elastic material (elastodynamics) is taken into account. This physical phenomena can be well described by the Lamé-Navier equation. To solve this equation numerically, the boundary element method is implemented. The resulting linear system of equations needs to be set up and solved several times, for different frequencies. The large, dense matrices appearing in this linear system of equations are computationally expensive.

To speed up the computational time, the matrix computation is done on GPU. The parallel processing capability of the Graphics Processing Unit (GPU) allows us to divide complex computing tasks into several thousands of smaller

** The final publication is available at
http://link.springer.com/chapter/10.1007%2F978-3-642-29843-1_39

tasks that can be run concurrently. We achieved an acceleration of $23 - 43$ in comparison to a computation performed on the CPU, serially.

2 Problem Setting

In an elastic body $\Omega \subset \mathbb{R}^3$ with a Lipschitz boundary $\Gamma = \Gamma_D \cup \Gamma_N$ and a fixed final time $T \in \mathbb{R}^+$ the following mixed initial boundary value problem has to be solved:

$$-(\mu+\lambda)\nabla\nabla.\mathbf{u}(\mathbf{x},t) - \mu\Delta\mathbf{u}(\mathbf{x},t) + \rho\frac{\partial^2\mathbf{u}}{\partial t^2}(\mathbf{x},t) = 0 \quad (\mathbf{x},t) \in \Omega \times (0,T) \quad (1)$$

$$\mathbf{u}(\mathbf{y},t) = \mathbf{g}_D(\mathbf{y},t) \quad (\mathbf{y},t) \in \Gamma_D \times (0,T)$$
$$\mathbf{t}(\mathbf{y},t) := \tau_y u(\mathbf{y},t) = g_N(\mathbf{y},t) \quad (\mathbf{y},t) \in \Gamma_N \times (0,T)$$
$$\mathbf{u}(\mathbf{x},0) = \tfrac{\partial\mathbf{u}}{\partial t}(\mathbf{x},t) \quad (\mathbf{x},t) \in \Omega \times (0)$$

The surface displacements $\mathbf{u}(\mathbf{x},t)$ and tractions $\mathbf{t}(\mathbf{x},t)$ are prescribed by some given data $\mathbf{g}_D(\mathbf{x},t)$ on Γ_D and $\mathbf{g}_N(\mathbf{x},t)$ on Γ_N, respectively. The traction operator τ_x reads as

$$(\tau_x\mathbf{u})(\mathbf{x},t) = (\sigma.\mathbf{n})(\mathbf{x},t) \quad (2)$$

with the stress tensor $\sigma(\mathbf{x},t)$ incorporating Hooke's law and the outward normal vector $\mathbf{n}(\mathbf{x})$ on the boundary Γ. The Lamé constants μ and λ are connected to the modulus of elasticity E and Poisson's ratio ν

$$\mu = \frac{E}{2(1+\nu)}, \quad \lambda = \frac{\nu E}{(1+\nu)(1-2\nu)} \quad (3)$$

whose physical significance is more immediate [3].

3 Numerical Schemes

The boundary element method is implemented on equation (1). To do the space discretization the collocation technique and for time discretization the Convolution Quadrature Method (CQM) are applied, and the following final discretized equation is achieved:

$$V(s_l)t_l - K(s_l)u_l = Cu_l \ . \quad (4)$$

The equation (4) consists of N elliptic problems for the complex 'frequency' sl, $l = 0, 1, ..., N-1$, where N is the total number of time steps.

The entries of the matrices $V(s_l)$ and $K(s_l)$ come from the following formulas:

$$V(s_l) = \sum_{e=1}^{E} \sum_{f=1}^{F} \int_{\Gamma_e} \hat{U}^*_{ij}(x, y, s_l) N_e^f(x) \, \mathrm{d}\Gamma_e \qquad (x \in \Gamma_e) \tag{5a}$$

$$K(s_l) = \sum_{e=1}^{E} \sum_{f=1}^{F} \int_{\Gamma_e} \hat{T}^*_{ij}(x, y, s_l) N_e^f(x) \, \mathrm{d}\Gamma_e \qquad (x \in \Gamma_e) \tag{5b}$$

where E is the number of boundary elements (Γ_e) on the mesh and F is the number of vertices belonging to each element. Here, the boundary elements are triangles. The $\hat{U}^*_{ij}(x, y, s_l)$ and $\hat{T}^*_{ij}(x, y, s_l)$ are the displacement and traction fundamental solutions, respectively. The fundamental solutions in this case are expressed as tensors. $N_e^f(x)$ is the basis function defined on each element. y is the collocation point. Γ_e is the boundary element where the integral is computed on. C is a diagonal matrix corresponding to the singularity appears when formulating the integral equations and all its entries are added to the diagonal elements of the matrix K.

The integrals appear in equations $(5a, 5b)$ must be solved numerically. All regular integrals are performed with Gaussian quadrature formulas. The strong singular integrals are performed with the method from [2] and the weak singular ones with polar coordinate transformation.

For simplicity of presentation, we restrict ourselves in the numerical example to the case of regular integrations. The implementation of the singular integrals will be similar with respect to GPU acceleration. We also restrict ourselves to the 79 number of Gauss points per element for presentation purposes.

After implementing the Gauss quadrature method on equations $(5a, 5b)$, we will have the following formulas:

$$V(s_l) = \sum_{e=1}^{E} \sum_{f=1}^{F} \sum_{g=1}^{G} \hat{U}^*_{ij}(x_g, y, s_l) N_e^f(x_g) \, W_g |Gram| \tag{6a}$$

$$K(s_l) = \sum_{e=1}^{E} \sum_{f=1}^{F} \sum_{g=1}^{G} \hat{T}^*_{ij}(x_g, y, s_l) N_e^f(x_g) \, W_g |Gram| \tag{6b}$$

here G, x_g, W_g and $|Gram|$ are the total number of Gauss points in each element, the Gauss point, the quadrature weight and the Gram determinant, respectively.

4 Parallel Computing Approach

In the large scale problems, where the same computations must be done several times and at the same time these computations are computationally expensive, the parallel computing can play a role to overcome these bottlenecks.

Parallel computing is a form of computation in which many calculations are carried out simultaneously, operating on the principle that large problems can often be divided into smaller ones, which are then solved concurrently ("in parallel"). Among different possible parallel computing approaches, here GPGPU programming is adopted.

For the purpose of GPU programming, the CUDA hardware and software architecture is implemented. The graphical card in this case is NVIDIA GeForce GTX 480.

4.1 Parallel Algorithm

To compute the matrices $V(s_l)$ and $K(s_l)$, appearing in the equations $(6a, 6b)$, on GPU, the 2D grid is taken into account. In this case, the thread block is also defined in 2D. If, as an example, the number of threads in each thread block is chosen as 64, 8 in each direction x and y, the number of thread blocks in the x-direction will be equal to the number of elements (on the geometrical mesh) divided by 8. The same is true for the number of thread blocks in the y-direction which is derived by splitting the number of collocation points into 8 equal parts. The grid configuration for this case is depicted in figure 1.

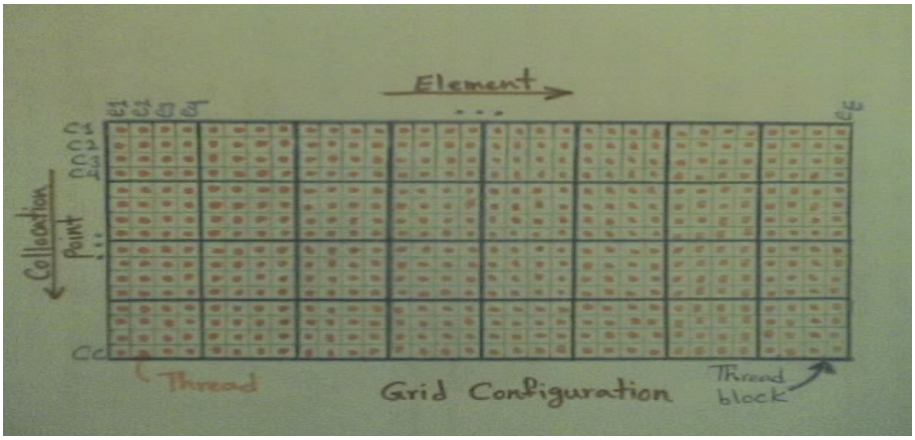

Fig. 1. The grid of thread blocks: it shows how the grid configuration looks like in the given parallel algorithm.

All these computations $\sum_{g=1}^{G} \hat{U}_{ij}^{*}(x_g, y, s_l) N_e^f(x_g) W_g |Gram|$ and $\sum_{g=1}^{G} \hat{T}_{ij}^{*}(x_g, y, s_l) N_e^f(x_g) W_g |Gram|$ appeared in equations $(6a, 6b)$ are done on each thread. In other words, all the computations corresponding to the pair combination of one element and one collocation point are performed on one thread. In addition, the same collocation point is used for all the thread blocks on the same row and to get access to the corresponding collocation point the

blockIdx.y indexing is used. The blocks on the same columns deals with the computations of the same element but different collocation points.

At the end, 3 number of 3×3 matrices are computed on each thread corresponding to each vertex of the same element (on the mesh). Then, these results must be stored in the correct place in the given matrices $V(s_l)$ and $K(s_l)$. To do that, the values computed on each thread corresponding to the same node on the geometrical mesh are summed up and stored in the position corresponding to that node in the given matrix on the global memory. The global memory has enough storage space, so the $V(s_l)$ and $K(s_l)$ matrices are stored there.

Since all the threads in the same thread blocks need to read the local coordinates of the Gauss points and the quadrature weight values repeatedly in a loop over the Gauss points and with respect to the fact that having access to the data stored on the shared memory is much more faster than the data on the global memory, to gain benefit out of it, all these values are stored in the shared memory .

To clarify what was explained earlier, part of the code is put here:

```
int main ()
{
  ...
//2D block -- 64 threads in each block
dim3 dimBlock (8,8);
//num_1: number of elements
//number of blocks in the x-direction
const int num_block_x = ceil(num_1/8);
//num: number of collocation points
//number of blocks in the y-direction
const int num_block_y = ceil(num/8);
dim3 dimGrid (num_block_x,num_block_y); //2D Grid

//on the device (GPU)
matrix_element<<<dimGrid,dimBlock>>>(...);

cudaThreadSynchronize();
...
return(0);
}
```

```
__global__ void matrix_element (...)
{
 #define L 79

 cuDoubleComplex U[9];
 double result, gram_y;
 cuDoubleComplex result_complex;
 int elementId, collocationId;
```

```
cuDoubleComplex sum[9];
......
//storing the Gauss points on the shared memory:
__shared__ double x[L*2];
for (int i=0; i<(L*2); i++)
  x[i] = *(x_d+i);

__shared__ double quad_weight[L];
for (int i=0; i<L; i++)
  quad_weight[i] = *(quad_weight_d+i);

for(int i=0; i<3; i++)
    {
     for(int j=0; j<3; j++)
       {
        sum[i*3+j] =  make_cuDoubleComplex(0.0,0.0);
        U[i*3+j] =  make_cuDoubleComplex(0.0,0.0);
         ......
       }
    }

//Loop over all Gauss points in each element
for(int l=0; l<L; l++)
  {
  //Evaluate the shape function:
  evaluateShapeFun (result, x, l);

  //Multiplying the weight function with the shape
  //function:
  result = result*quad_weight[l]*(0.5);

  //Gram determinant --- is the mapping function
  //from the reference element to the real element:
  Gram_determinant (gram_y, ...);

  //Multiplying the result with the gram determinant:
  result *= gram_y;

  //change the result from double to cuDoubleComplex:
  result_complex = make_cuDoubleComplex(result,0.0);

  //Single layer fundamental solution
  //Tensor with 9 elements in 3D---U
  //x: Gauss point
  evaluateFundSol_SLP (U, ..., x, l);
```

```
    //Multiplication of all the elements taking part into
    //the computations. sum is the matrix element
    //corresponding to each node and collocation point:
    for(int i=0; i<3; i++)
      {
       for(int j=i; j<3; j++)
         sum[i*3+j] += cuCmul(U[i*3+j],result_complex);
      }

    sum[3] = sum[1];
    sum[6] = sum[2];
    sum[7] = sum[5];
    __syncthreads();
    ......
    }

  //A is the final square matrix on global memory.
  //Plug the value of sum into the A matrix.
  //The value of the same node/vertex on different threads
  //summed up together and insert into the corresponding
  //matrix element:
  elementId = (threadIdx.x+1)+(blockDim.x*blockIdx.x);
  elementId = connectivity_matrix_d[(elementId-1)*3];
  collocationId = (threadIdx.y+1)+(blockDim.y*blockIdx.y);
  elementId = (elementId-1)*3;
  collocationId = (collocationId-1)*3;
  collocationId = (collocationId*3*(*N))+elementId;
  for(int i=0; i<3; i++)
    {
     for(int j=0; j<3; j++)
       A[collocationId+(3*(*N)*j)+i] += sum[i+(3*j)];
    }
   __syncthreads();
   .....
};
```

5 Numerical Examples

Equation (1) is solved for the special case where a steel rod, which is fixed from one end and free from the other side, is excited by pressure P in its longitudinal direction. The code is run for 10 number of time steps and $\Delta t = 0.01$ on both CPU and GPU, separately and the elapsed time for each case corresponding to different number of mesh elements is measured. These values are tabulated in table 1.

Table 1. Elapsed time (seconds)-Number of time steps= 10, $\Delta t = 0.01$

Number of elements	CPU	GPU
12	0	0.00231
112	0.22	0.00512
822	7.66	0.275237
3288	103.97	4.46914

6 Conclusion

As it can be seen in table 1, the computations done on GPU are $23 - 43$ times faster than the CPU ones. We tried to increase this ratio $\frac{number\ of\ arithmetic}{memory\ transfer}$, by performing all the computations corresponding to one element and one collocation point on one thread and putting all those data used repeatedly by all threads of the same thread blocks in the shared memory.

In this case, the number of threads in each thread block is chosen to be 64, but it is going to be increased to 128 and 256, respectively.

We expect to improve the acceleration of the code further, by using the registers and other fast access memory (shared memory) in more efficient way and at the same time looking for the best distribution of the computations among the available threads and thread blocks. We are going to see whether all these changes can improve the acceleration any further more.

References

1. Schanz, M.: Wave Propagation in Viscoelastic and Poroelastic Continua: A Boundary Element Approach, volume 2 of Lecture Notes in Applied Mechanics (2001)
2. Guiggiani, M., Gigante, A.: A general algorithm for multidimensional cauchy principal value integrals in the boundary element method. J. of Appl. Mech. **57** (1990) 906–915.
3. Schanz, M.: On a Reformulated Convolution Quadrature Based Boundary Element Method. Computer Modeling in Engineering and Sciences. **58** (2010) 109–128.
4. NVIDIA Programming Guide Documents