Engineering Analysis with Boundary Elements 74 (2017) 34–48

Contents lists available at ScienceDirect

Engineering Analysis with Boundary Elements

journal homepage: www.elsevier.com/locate/enganabound

GPU acceleration of the boundary element method for shear-deformable bending of plates

Ahmed A. Torky[a,b], Youssef F. Rashed[b,c,*]

[a] Department of Civil Engineering, The British University in Egypt, Cairo, Egypt
[b] CUFE-BE Research Group: http://eng.cu.edu.eg/postgraduate-portal/research-groups, Department of Structural Engineering, Cairo University, Giza, Egypt
[c] Supreme Council of Universities, Egypt

ARTICLE INFO

Keywords:
CUDA
Boundary element method
Reissner plate
GPU
Parallel computing
PLPAK

ABSTRACT

This paper presents a novel implementation of GPU computing for boundary element method (BEM) formulation of plates. The new GPU code written in CUDA Fortran alters three kernels: the calculation of influence matrices, the solution of equations, and the computation of internal values. Comparisons for computation time and different GPU architecture are presented. The formulation is implemented for both constant and quadratic elements. The efficiency of the parallel quadratic code is demonstrated via analysis of practical building slabs. The benefits of parallel computing of the solution of the system of equations and of points inside the domain are discussed.

1. Introduction

In computation mechanics, the use of the Graphics Processing Unit (GPU) instead of the Central Processing Unit (CPU) for bulk-processing of data has become a trend since 2009. The GPU is now investigated by many scientists to develop high performance computing (HPC) of numerical computational methods, in order to improve the efficiency of the Boundary element method (BEM) [1], Finite element method [2], N-body problems [3], and Finite difference method [4]. The use of the GPU to process immense and independent data has eliminated the archaic use of sequential processing and kick started new parallel techniques.

The GPU technology was originally developed to target graphics, image processing, etc.. Seeing a need in the scientific community, NVIDIA Corporation launched a parallel computing platform and programming environment readily available for scientists in several languages (C, Fortran, Python) to utilize the GPU. The GPU is a scalable computing tool proficient in executing a tremendous number of threads independently in parallel. Directed as a coprocessor to the CPU, data could be off-loaded on to the GPU, computed and the results transferred back to the CPU. This API from NVIDIA is called Compute Unified Device Architecture (CUDA) [5], which is used to employ parallelism, complements the used languages in syntax and easy to use. Another parallel computing environment is OpenCL (Open Computing Language) [6] which enables the user to make use of multiple platforms. Although OpenCL is similar to CUDA, the former lacks the substantial development that the latter has, e.g. online linear algebra libraries and implementations in computational mechanics.

Graphics hardware technologies from NVIDIA are parallel computation tools that contain thousands of “CUDA Cores” on a single device. To achieve the same level of parallelism with CPUs, one would need many units to produce a cluster of cores comparable with cores handled by only a single GPU. The GPU is therefore much more attractive for large scale processing due to affordability of one GPU over a cluster of CPUs.

Two of the most common criteria for assessing processing units are latency and throughput. According to the size of the data to be processed, one must consider if the completion of one piece of data needs to be essentially fast (low latency) or the rapid completion of all the data computations (high throughput) needs to be the focus. The GPU (device) has higher latency but, due to efficient parallelism, high throughput and vice versa for the CPU (host). This gives the programmer the choice of solving small sized data on the low latency host while leaving a bulky data to be solved on the device.

Since NVIDIA released CUDA, several large scale computational methods have utilized this paradigm shift in computer architecture. A computational technique for a discretized boundary using the boundary integral equation is the BEM. The theories and the algorithms of

* Corresponding author at: CUFE-BE Research Group: http://eng.cu.edu.eg/postgraduate-portal/research-groups, Department of Structural Engineering, Cairo University, Giza, Egypt.
E-mail addresses: ahmed.torky@bue.edu.eg (A.A. Torky), youssef@eng.cu.edu.eg, yrashed@scu.eg (Y.F. Rashed).

http://dx.doi.org/10.1016/j.enganabound.2016.10.006
Received 1 June 2016; Received in revised form 4 September 2016; Accepted 7 October 2016
Available online 02 November 2016
0955-7997/ © 2016 Elsevier Ltd. All rights reserved.

BEM in continuum mechanics are a prime target for parallel programming during its pre-processing and the post-processing stages. These processes mainly contain numerical integrations in a dense linear system populated by influences of the boundary elements to one another. Takahashi and Hamada [1] were among the first to use the GPUs to produce an accelerated 3D Helmholtz BEM program using GeForce GX8800GTS. Labaki et al. [7] presented a parallel constant BEM solution for 2D potential problems using GeForce GTX 280 GPU; whereas Yingjun et al. [8] presented a quadratic BEM parallel computation method for 3D elastostatics on GeForce GTX 285 GPU. Other recent applications were done: improving elastostatic analysis on GPUs using Tesla C2075 [9]; fast multipole BEM for biomolecular electrostatics [10] and for3D elasticity [11]. All applications showed the superiority of the GPU over the CPU in throughput of large scale data with CUDA enabled software reaching 40 times the speed of the equivalent CPU serial program.

In this paper, CUDA Fortran programming [12] is applied to the boundary element method of Reissner's plate bending theory [13]. A description is presented of how the boundary element method is broken up to three kernels that are off-loaded on to the GPU, which are: the calculation of the influence coefficients, the solution of the dense linear system and the processing of internal points inside the domain. The order of degrees of freedom of each element's boundary conditions is O(3N) whereas the influence matrices are $O(3N)^2$, making them suitable for the GPU architecture. The results are compared to traditional CPU results.

The development is firstly concerned with constant elements, and then extended to deal with quadratic elements with the help of the previously developed commercial code of the PLPAK. Practical building slabs are then tested to demonstrate the efficiency of the proposed GPU computing.

The following sections discuss: Section 2. the shear deformable plate bending theory in more details, Section 3. the CUDA's structure, Sections 4–6. the development of GPU Constant element code, Section 7. a parametric study on a selected model to compare performances between the GPU's and CPU's codes, Section 8. the proposed development of a GPU Quadratic element code inside the PLPAK, Section 9. the results of the GPU Quadratic element code on practical building slabs and foundations, and lastly Section 10. the conclusions.

2. BEM for plate bending

The boundary element method [14] is a versatile tool in structural engineering analysis. The basic BEM for shear deformable plates was developed by Vander Weeën [15–20]:

$$C_{ij}(\xi)u_j(\xi) + \int_{\Gamma(x)} T_{ij}(\xi,x)u_j(x)d\Gamma(x) = \int_{\Gamma(x)} U_{ij}(\xi,x)t_j(x)d\Gamma(x)$$
$$+ \int_{\Gamma(x)}\left[V_{i,n}(\xi,x) - \frac{\nu}{(1-\nu)\lambda^2}U_{i\alpha}(\xi,x)\right]q d\Gamma(x)$$
$$+ \sum\nolimits_{cells}\int_{\Omega_{cells}}\left[U_{i\,k}(\xi,c) - \frac{\nu}{(1-\nu)\lambda^2}U_{i\alpha,\alpha}(\xi,c)\delta_{3k}\right]q_k(c)\,d\Omega_c(c)$$
$$+ \sum\nolimits_{columns}\int_{\Omega_{col.}}\left[U_{i\,k}(\xi,y) - \frac{\nu}{(1-\nu)\lambda^2}U_{i\alpha,\alpha}(\xi,y)\delta_{3k}\right]F_k(y)\,d\Omega_{col.}(y) \tag{1}$$

where $U_{ij}(\xi,x)$ and $T_{ij}(\xi,x)$ are the fundamental solution kernels in Vander Weeën [15], $u_j(x)$, $t_j(x)$ are the boundary displacements and tractions, $C_{ij}(\xi)$ is the jump term. The terms *cells* and *columns* are the number of internal beam cells and columns which have domains Ω_{cells} and $\Omega_{col.}$ respectively, F_k and q_k denote the columns two bending moments and column vertical force, whereas field points c and y denote the point of the internal beam cells and column centers respectively. Beams and any other supports are treated as columns with different stiffness values.

The corresponding matrix is

$$[H]_{3N\times3N}\{u\}_{3N}=[G]_{3N\times3N}\{p\}_{3N}+\{D\}_{3N} \tag{2}$$

where $[H]$ and $[G]$ are the boundary element influence matrices, $\{D\}$ is the domain load vector, $\{u\}$ and $\{p\}$ are the vectors of the boundary displacements and tractions. The boundary solution is obtained through the solution of

$$[A]_{3N\times3N}\{x\}_{3N}=\{b\}_{3N} \tag{3}$$

where$[A]$ is the grouping of the $[H]$ and $[G]$ columns related to the unknown boundary conditions $\{x\}$, $\{b\}$ is the product of $[H]$ and $[G]$ columns and their known boundary conditions, whereas N is the number of unknowns either on the boundary and/or as internal supporting elements [16–19]. The two discretization types of the boundary in the present work are constant and quadratic, as will be discussed late in Sections 4 and 8.

Once the boundary solution is obtained, values of the generalized displacements and stress resultants inside the domain could be calculated. Generalized displacements for any internal point ξ can be obtained using Eq. (1) with $C_{ij}(y)=\delta_{ij}$. The stress resultants at an internal point ξ are obtained as follows [15]:

$$M_{\alpha\beta}(\xi) = \int_{\Gamma(x)} U_{\alpha\beta k}(\xi,x)t_k(x)d\Gamma(x) - \int_{\Gamma(x)} T_{\alpha\beta k}(\xi,$$
$$x)u_k(x)d\Gamma(x) + q\int_{\Gamma(x)} W_{\alpha\beta}(\xi,$$
$$x)d\Gamma(x) + \frac{\nu}{(1-\nu)\lambda^2}q\delta_{\alpha\beta} + \sum_{cells\ \&\ columns}\int_{\Omega_{c.}}\Bigg[U_{\alpha\,\beta\,k}(\xi,$$
$$c) - \frac{\nu}{(1-\nu)\lambda^2}U_{\alpha\,\beta\,\theta,\theta}(\xi,c)\delta_{3k}\Bigg]q_k(c)\,d\Omega_c(c) \tag{4}$$

$$Q_{3\beta}(\xi) = \int_{\Gamma(x)} U_{3\beta k}(\xi,x)t_k(x)d\Gamma(x) - \int_{\Gamma(x)} T_{3\beta k}(\xi,$$
$$x)u_k(x)d\Gamma(x) + q\int_{\Gamma(x)} W_{3\beta}(\xi,x)d\Gamma(x) + \sum_{cells\ \&\ columns}\int_{\Omega_{c.}}\Bigg[U_{3\,\beta\,k}(\xi,$$
$$c) - \frac{\nu}{(1-\nu)\lambda^2}U_{3\,\beta\,\theta,\theta}(\xi,c)\delta_{3k}\Bigg]q_k(c)\,d\Omega_c(c) \tag{5}$$

where the new kernels $U_{ijk}(\xi,x)$ and $T_{ijk}(\xi,x)$ and their derivatives are given in Appendix A. Any internal support of any kind is converted in to a loading cell with known loading conditions.

3. GPU parallel processing

Processing of the plate bending problem is transferred to the GPU hardware to utilize its parallel performance capabilities. The following subsections will recount the various GPU programming tools that were used in the current study.

3.1. Architecture overview

Currently, all modern NVIDIA GPUs [4] architectures are many-core processors that support CUDA. The architecture is simplified for scientists and engineers to understand, in order to efficiently use the GPU. A GPU has a global memory and is broken up to several Streaming Multiprocessors (SMs). SMs control a number of CUDA Cores, with the quantity of each differing for each model and usually increasing as the technology advances. Cores are small streaming processors that are managed by the SM, whereas each core itself processes a warp at any given time. A warp is an instruction of 32 instances, termed threads, which are computed concurrently on one core.

Threads are where each independent computation (single instruction) on the GPU takes place in parallel. Threads are uniquely identified in their parent thread block (a.k.a. block) using indices (threadIdx), explained clearly in [7]. Thread blocks could contain an array of threads, with the array dimensions either 1, 2 or 3. Thread blocks are

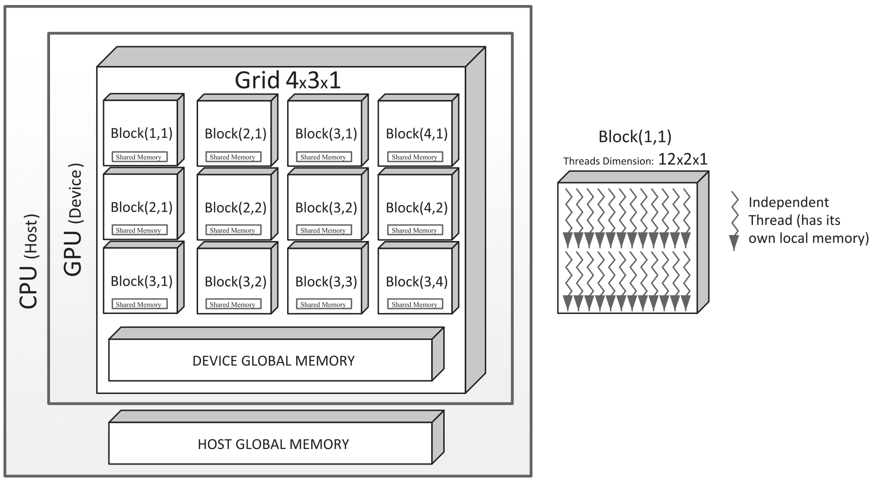

Fig. 1. An example of the hierarchy of the device. The dimensions of the Grid and Block in this case are revealed to be (4×3×1) and (12×2×1) respectively.

similarly perceived in their parent Grid also using indices (blockIdx). Grids also contain an array of thread blocks, with the array dimensions either 1, 2 or 3. Thus the grid is a set of thread blocks that execute a GPU operation set by the CPU, meanwhile thread blocks count and dimensions could be optimized to compute a set of single instruction tasks efficiently without calling extra void thread blocks and threads. This logical hierarchy is demonstrated in an example in Fig. 1 and is directly managed by the programmer through CUDA, as will be explained in Section 3.2.

Newer GPUs usually have more CUDA cores which are more efficient [21]. This allows the GPU to directly achieve lower latency and higher throughput relative to older GPUs. Each SM will be occupied by one thread block at a time. A GPU that has more SMs will run more blocks concurrently due to dynamic scheduling. It is therefore preferable that a large number of blocks are composed for a grid to allow for scalability consistent with the model of the NVIDIA GPU.

3.2. CUDA Fortran

CUDA Fortran is a product brought by the collaboration of Portland Group [22] and NVIDIA [23]. It enables CUDA programming in Fortran through a concise set of extensions that directly supports and manages information on the computing architecture. Through a thoroughly written, and CUDA-enabled, Fortran code the program compiled has the potential to: (1) Declare variables that populate diverse device memories, (2) Efficiently allocate data on the device, (3) Transfer data from host to device and vice versa, (4) Process CUDA enabled modules and subroutines (device kernels) from the host, (5) Call commercial CUDA libraries (e.g. accelerated linear algebra).

CUDA Fortran kernels execute in parallel on the device with the size and dimensions of the grid and blocks coordinated by the host. With a thread being the elementary unit of the hierarchy inside the grid, each instance of the kernel is executed by all the CUDA threads.

A program that targets high throughput on the GPU with CUDA (targets optimization) should at least focus on transforming a sequential subroutine to a parallel kernel and eliminate unnecessary data transfers between host and device. It is necessary to regulate kernel initiation to be of the best configuration for GPU architecture in regards to block dimensioning and grid dimensioning (occupancy ratio).

The Host program (main) is written with the device subroutines' calls defined in the form:

call SubroutineName« < GridDim, BlockDim > > > (DeviceVariables).

where **GridDim** and **BlockDim** are defined as **type(dim3)**, and their purpose is to coordinate the division of the Grid and the containing blocks in at most three dimensions.

e.g. BlockDim=dim3(16×16×1), GridDim=dim3(N/16,N/16,1).

The device subroutine, labeled as the device kernel, is called:

attributes(global) subroutine SubroutineName(DeviceVariables).

and the remainder of the subroutine is similar to the serial CPU code which does the same process, with the exception of the elimination of nested loops. The same device kernel is run on each thread inside each block, and the only current distinction between all threads is identified by the thread's index and the block's index in the following manner (in the case of a three dimensional block with a three dimensional grid):

```
I=threadIdx%x+(blockIdx%x-1) * blockdim%x
J=threadIdx%y+(blockIdx%y-1) * blockdim%y
K=threadIdx%z+(blockIdx%z-1) * blockdim%z
```

These variables are assigned to read all the indices in 'x,y,z' directions of the threads (threadidx%...), block dimensions (blockdim %...), and blocks contained by grid (blockidx%...). Thus, all the variable arrays that a thread should operate on could be a function of 'I,J,K' in order to distinguish between different instances of computation. The device kernel's lines of code progress with sequential order from launch to end on each thread and there would be no parallelism internally on this serial part. As will be described in more details later, this easy form of the dispersal of computation condenses 'do loops' that are thrice nested and simplifies addition reduction of results.

Operations on a thread could be divided into arithmetic and memory operations. Arithmetic instructions are preferable as they have lower latencies than memory instructions, thus the GPU code proposed here has the least amount of memory instructions by all threads possible to increase device kernel performance.

There are three major parts in boundary element analysis of slabs that require parallelism through CUDA, and those are the assembly of elements' influences on each other, the solution of the boundary values and the post-processing of internal points. The following sections will describe each part in detail, which is represented in the flow diagram in Fig. 2.

4. Matrix assembly of influence coefficients ([H] & [G]) for constant elements

In this section, matrix assembly of the two influence matrices $[H]$ and $[G]$ are considered on to the GPU. It is sensible to have each influence of a source point on to a field element mapped on to a thread, and allow the 3×3 sub-matrix to be formed by the device subroutine serially. This is more suitable for calculation time, as many preparatory

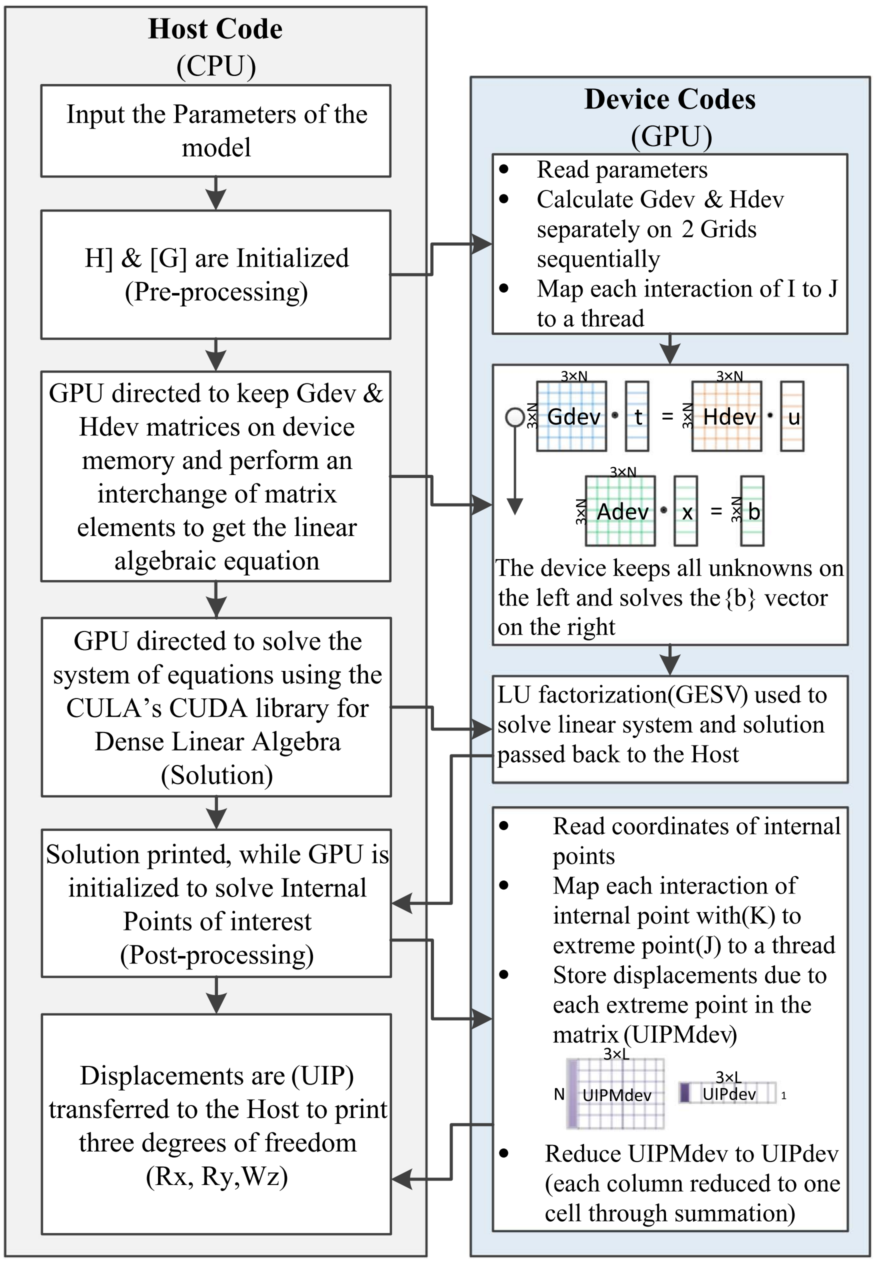
Host Code (CPU)
Input the Parameters of the model
H] & [G] are Initialized (Pre-processing)
GPU directed to keep Gdev & Hdev matrices on device memory and perform an interchange of matrix elements to get the linear algebraic equation
GPU directed to solve the system of equations using the CULA's CUDA library for Dense Linear Algebra (Solution)
Solution printed, while GPU is initialized to solve Internal Points of interest (Post-processing)
Displacements are (UIP) transferred to the Host to print three degrees of freedom (Rx, Ry,Wz)
Device Codes (GPU)
Read parameters
Calculate Gdev & Hdev separately on 2 Grids sequentially
Map each interaction of I to J to a thread
3×N
Gdev
t
Hdev
u
Adev
x
b
The device keeps all unknowns on the left and solves the {b} vector on the right
LU factorization(GESV) used to solve linear system and solution passed back to the Host
Read coordinates of internal points
Map each interaction of internal point with(K) to extreme point(J) to a thread
Store displacements due to each extreme point in the matrix (UIPMdev)
3×L
N
UIPMdev
UIPdev
1
Reduce UIPMdev to UIPdev (each column reduced to one cell through summation)

Fig. 2. The flow of data in the new hybrid code for the constant GPU code.

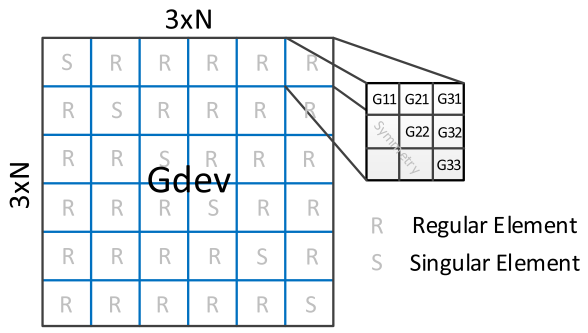

Fig. 3. A schematic diagram for **Gdev**.

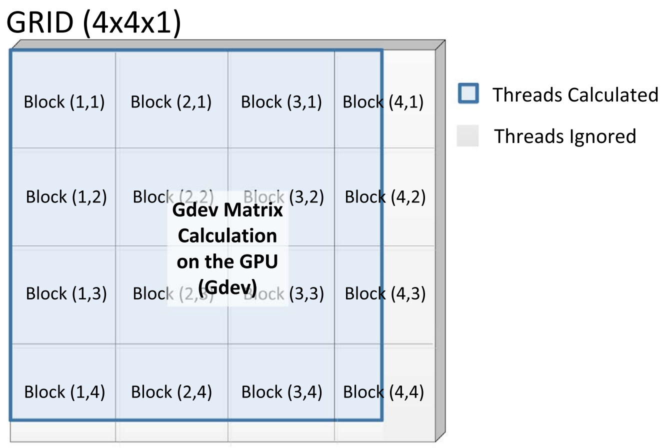

Fig. 4. The grid maintains the whole size of **Gdev**. With a logical statement extra threads are not executed.

calculations are carried out before solving each part of the sub-matrix individually whereas all these calculations are mainly unified for each element to element influence. In the serial code, the $[H]$ and $[G]$

matrices are of size $O(3N)^2$, therefore they will be transformed into a grid that is of size $O(N^2)$. Each influence matrix had its components calculated through three nested loops, and in the most internal loop Gauss' numerical integration is considered which further increased complexity of the serial code. Such components are no longer in nested loops in the device subroutine, with the Gauss' numerical integration being the only loop remaining. Algorithm 1 gives a demonstration of the CUDA Fortran code part for the calculation of the **Gdev** matrix, which simulates the $[G]$ matrix on the device.

Algorithm 1. Host code and Device kernel for **Gdev** matrix computation.

```
Program Plate_Bending_CUDA_Fortran
use cudafor
  …
call CPU_Time(Time0)
call GMATGPU« < dim3(real(N/16)+1, real(N/16)+1,1),
  dim3(16,16,1) > > > (X,Y, Gdev,N, NX)
istat = cudaDeviceSynchronize()
call CPU_Time(Time1)
print(*,*) 'Time spent to calculate Gdev Matrix: ', Time1-Time0
  …
End
Attributes(global) Subroutine GMATGPU(X,Y,G,N, NX)
  {allocate all variables}
  Integer I, J, F, i1, i2, i3, j1, j2, j3
  Integer, Value:: N, NX
  Real:: G(NX, NX), XP, YP, X(N), Y(N), G11, G12, G13, G22,
  G23, G33
  {identify thread parameters}
  I= threadIdx%x + (blockIdx%x−1) * blockdim%x
  J= threadIdx%y + (blockIdx%y−1) * blockdim%y
  i1=3*I−2,i2=3*I−1, i3=3*I, j1=3*J−2, j2=3*J−1, j3=3*J
  {initialize local variables G11, G12, G13, G22, G23, G33}
If (I  < = N.and. J  < =N) then
If (I.EQ. J) then
  {calculate coefficients by numerical integration and analytical
  integration}
  {calculate the regular part by numerical integration: 10 gauss
  points for precision}
  Do F=1,10
  {calculate bessel functions K0,K1}
  {calculate G11, G12, G22}
  End Do
{calculate remaining coefficients by analytical integration only
  G13, G23, G33}
Else {I.NE. J source and field point are on the same element}
  Do F=1,10
  {calculate bessel functions K0,K1}
  {calculate coefficients by numerical integration only G11, G12,
  G13, G22, G23, G33}
End Do
End If
End If
{populate G(NX, NX) with computed coefficients}
G(i1,j1)=G11, G(i1,j2)=G12, G(i1,j3)=G13, G(i2,j1)=G12,
  G(i2,j2)=G22
G(i2,j3)=G23, G(i3,j1)=−G13, G(i3,j2)=−G23, G(i3,j3)=G33
End Subroutine GMATGPU
```

Focusing on the $[G]$ kernel (Fig. 3), it can be seen that for each source point ξ to field element x, nine influence coefficients are need to be calculated. The kernel is distributed on to the grid, to solve each chunk of the nine coefficients on to a single thread. An IF statement is activated when ξ=x, as singular terms need to be processed. Another IF statement (I < =N.and. J < =N), where I and J are thread identifiers, prevents extra threads from being launched (Fig. 4), as the dimension of the grid will always be greater than or equal to the size of the matrix being formed. Once the GPU completes the $[G]$ kernel (stored in **Gdev** array), commands proceed serially on the CPU to either transfer data from the global memory of the device to the host, or maintain the $[G]$ matrix on the device. It is more time efficient to keep necessary data on the GPU if it will be further processed in order to minimize the time wasted in data transfer. Assigning shared memory to variables such as the **Gdev** matrix is not necessary as the time spent for populating the matrix in the global memory is negligibly small compared to other calculations on the device.

The GPU code then initiates the device to calculate the $[H]$ kernel.

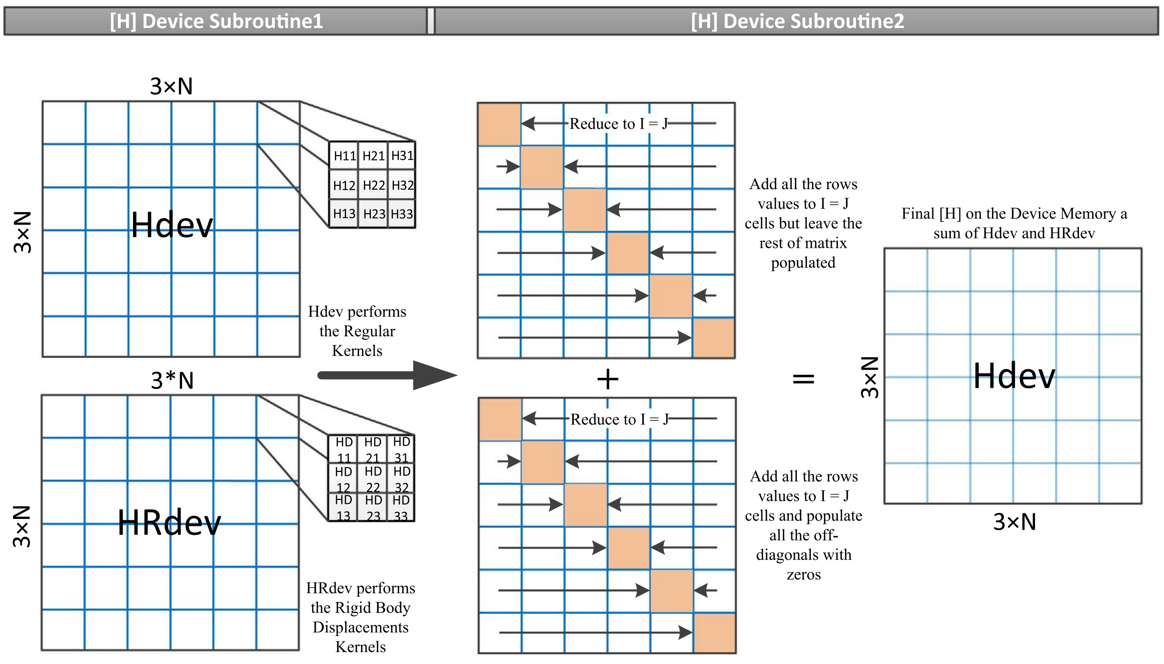
[H] Device Subroutine1
[H] Device Subroutine2
3×N
3×N
Hdev
H11 H21 H31
H12 H22 H32
H13 H23 H33
Hdev performs the Regular Kernels
3*N
3×N
HRdev
HD 11 HD 21 HD 31
HD 12 HD 22 HD 32
HD 13 HD 23 HD 33
HRdev performs the Rigid Body Displacements Kernels
Reduce to I = J
Add all the rows values to I = J cells but leave the rest of matrix populated
+
Reduce to I = J
Add all the rows values to I = J cells and populate all the off-diagonals with zeros
=
Final [H] on the Device Memory a sum of Hdev and HRdev
3×N
Hdev
3×N

Fig. 5. The **Hdev** matrix is computed by two device kernels (subroutines).

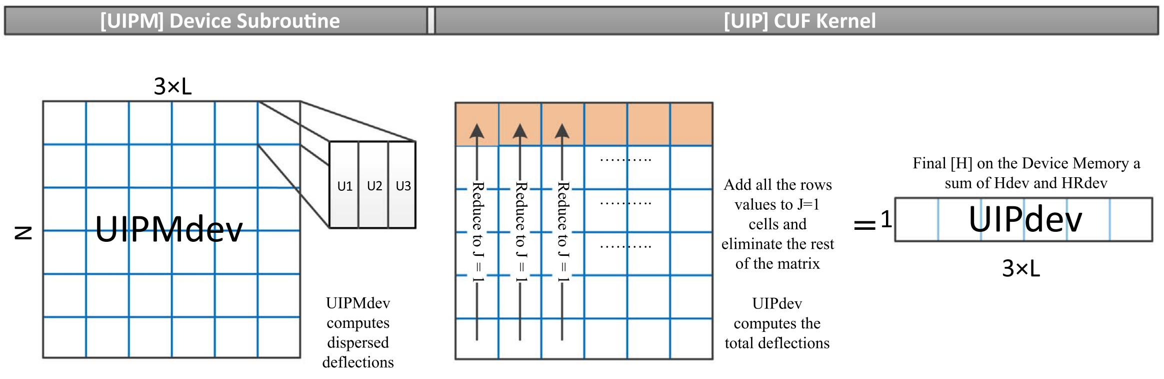
[UIPM] Device Subroutine
[UIP] CUF Kernel
3×L
N
UIPMdev
U1
U2
U3
UIPMdev computes dispersed deflections
Reduce to J = 1
Add all the rows values to J=1 cells and eliminate the rest of the matrix
UIPdev computes the total deflections
Final [H] on the Device Memory a sum of Hdev and HRdev
=1
UIPdev
3×L

Fig. 6. Calculation of the **UIPdev** through a device kernel and reduction through summation.

The $[H]$ kernel is much more intensive (Fig. 5), as the coefficients need to be processed twice to consider rigid body displacements. The first kernel will process the regular coefficients (stored in **Hdev** array) and the rigid body displacement coefficients (stored in **HRdev** array) independently. The second device kernel will utilize the two arrays on the device's global memory to reduce through summation the **Hdev** and **HRdev** arrays. All rows are summed and added to the diagonal and all the off-diagonals are kept as is (for **Hdev**) or leveled to zero (for **HRdev**). Then the second device kernel will progress with adding the cells of the array **HRdev** to **Hdev** to complete the $[H]$ matrix. This influence matrix is done on two steps to spread out the compound looping of the serial program to parallel independent procedures. These kernels minimize any unnecessary serial procedures and increase memory access efficiently on the device threads.

The host will then direct the GPU to build a system of equations that attain values from the coefficient matrices of **Hdev** and **Gdev**, and transfer unknown boundary conditions' coefficients to $[A]$ and compute the right hand vector $\{b\}$. This final part is simply a rearrangement of coefficients to store the system of equations on the device and solve it through LU decomposition in one go without excessive transfers back to the host.

5. LU decomposition for solution of system of equations

The $[A]$ matrix is dense, and thus will require dense linear algebra to process. LU decomposition is chosen to solve $[A]\{x\} = \{b\}$ on the device through the use of the "CULA Dense" linear algebra library [24] available as a free edition. "CULA Dense" is a pre-compiled implementation of linear algebra routines that are based on the BLAS/LAPACK package but tuned for graphic processing units that are CUDA-enabled. Massively intensive linear algebra procedures are computationally excessive and require $O(N^3)$ operations and thus $O(3N)^3$ in our application, which is practically suitable on the GPU. The unknown boundary conditions are computed from factorizations such as LU decomposition using the library termed 'sgesv' (single precision) and 'dgesv' (double precision) routines. The routines process data already on the device, which are $[A]$ and $\{b\}$, as seen in Algorithm 2. Both single and double precision values were coded to identify the accuracy of the results of the $\{x\}$ vector and the time performance of each precision on the same GPU.

Algorithm 2. Calling the Linear Algebra library CULA Dense to solve the linear system of equations.

```
Program Plate_Bending_CUDA_Fortran
use cudafor
  …
status = cula_initialize()
status = cula_dgesv(n,1,A,n,ipiv,x,n)
  …
End
```

6. Internal point independency

Once all the boundary values are computed, any internal point could be solved to acquire generalized displacements and stress resultants. However, any internal point that will be solved won't have any sequential relation with fellow points inside the domain; thus each point is entirely independent. For simplification, only displacements will be explained as the stress resultants follow exactly the same serial to parallel code transformation steps. The boundary displacements are assigned an array on the device and loaded from the now known vectors computed earlier.

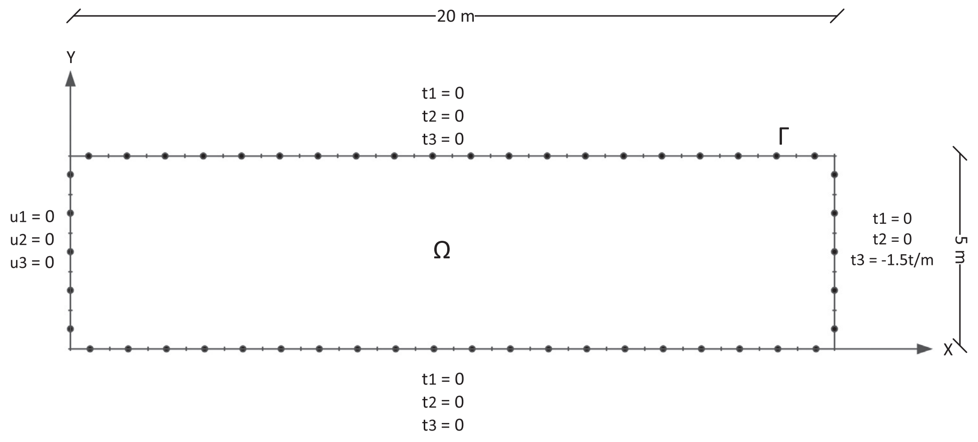

Fig. 7. The rectangular plate problem subjected to uniform constant gravitational load on one side and fixed on the opposite site.

Table 1
Hardware used for implementation.

Hardware	Specifications	Theoretical (Peak) GFLOPS Single Precision
CPU	Intel Core i7-3770 CPU @ 3.40 GHz with 16.0 GB memory	–
GPU (Maxwell)	NVIDIA Geforce GTX 970 with 13 SMs, 1664 CUDA Cores, 1216 MHz, 4.0 GB memory	3494
GPU (Kepler)	NVIDIA Geforce GTX 770 with 8 SMs, 1536 CUDA Cores, 1046 MHz, 2.0 GB memory	3213
GPU (Fermi)	NVIDIA Geforce GT 525 M with 2 SMs, 96 CUDA Cores, 600 MHz, 1.0 GB memory	230.4

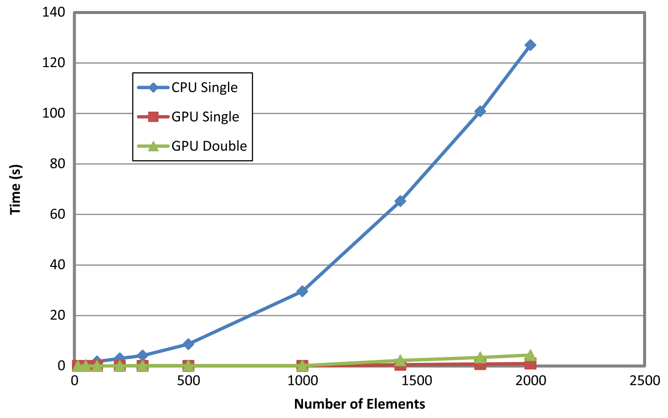

Fig. 8. A time comparison of influence matrices computation between CPU code and GPU code.

The influence of all boundary elements are calculated on each internal point (Fig. 6), thus a matrix **UIPMdev** could be formed on the device where a column represents the internal point numbering and the rows represent each boundary element's influence. The columns are further divided into 3 each, as the fundamental solution for displacements has 3 DOF inside the domain. When the device kernel is launched for the Internal Point subroutine, each thread computes a part of **UIPMdev**. IF statements distinguish between internal point number and boundary element number, whereas three displacements per each collocation are computed as a chunk together to reduce initiation time.

UIPMdev is then kept on the device and the array's columns are reduced through summation to obtain the final displacements (**UIPdev**) of each internal point. A simple reduction of such level is done through CUF Kernel Loop Directives [25]. The directive appears before tightly nested loops, and commands the compiler to transform the loops to device kernels, and gives the thread block dimension and grid size. The summation reduction for the **UIPMdev** to **UIPdev** is written using this kernel directive as shown in Algorithm 3. The block sizes are (1×1024), the grid size is (1xL/1024), (L) is the number of internal points and the '(1)' defines the level at which loops will be parallel (which will be the first loop only in this case). This ensures that no thread will duplicate calculation of another thread and memory accessibility will be coalesced. The loops will be executed in parallel, across the thread blocks and grid; the compiler handles the production of the final reduction kernel, deploying synchronization into the kernel as suitable. The **UIPdev** vector is then transferred to the host's memory in the array {UIP}.

Algorithm 3. Sum-Reduction of the **UIPdev** vector on device **UIPMdev**.

```
! $cuf kernel do (1) « <  L/1024, 1024  > > >
   Do I=1,L
   Sum1=0
   Sum2=0
   Sum3=0
   Do J=1, N
   Sum1=UIPMdev(3*I−2,J)+Sum1
   Sum2=UIPMdev(3*I−1,J)+Sum2
   Sum3=UIPMdev(3*I,J)+Sum3
   End Do
UIPdev(3*I−2)=sum1
UIPdev(3*I−1)=sum2
UIPdev(3*I)=sum3
End Do
UIP = UIPdev
Print(*,*) 'UIP: ', UIP
```

7. Numerical results for constant elements

The current implementation is deployed to solve the problem demonstrated in (Fig. 7) with the shown boundary conditions. The plate is basically fixed on one side and a uniform constant load is applied to its edge on the opposite side, whereas two other sides are left free. The plate is rectangular (5 m×20 m) and the shorter sides are where either the fixation or the loads reside. Properties of the plate are unified at: plate thickness is 0.3 m, modulus of elasticity is 3.0×10^7 kN/m^2 and the Poisson ratio is zero. The line load is 15kN/m perpendicular to the plate's surface and facing down.

The boundary elements are distributed with equal lengths along the whole boundary and several arrangements where solved ranging from 16 elements to 5000 elements to test the code performance. Ten Gauss points were calculated for numerical integrations using Gauss–Legendre Quadrature method. All the arrangements of the problem where computed using a serial "CPU code" and then using a device optimized "GPU code". Both processed data in exactly the same manner and with reliable accuracy. The results produced by each code were computed initially using single precision variables in the code, then the whole process is repeated using double precision variables. No deterioration of results accuracy is noticed. The hardware used is shown in Table 1.

Table 1 shows the three GPUs made available to the authors in the current study. Their processor power, CUDA cores and total memory is evident in the table, in addition to their theoretical performance at peak

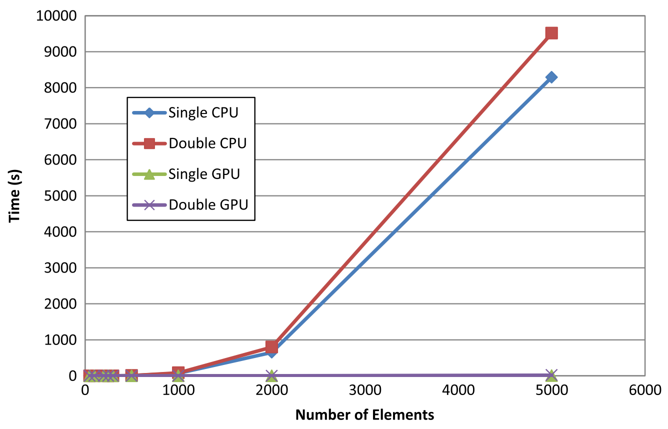

Fig. 9. A time comparison of LU decomposition solution between CPU code and GPU code.

hardware operation. Each GPU was manufactured by NVIDIA at different eras of architecture development. CUDA was first available on the Fermi architecture and then followed by Kepler and the most recent Maxwell architectures. Old GPUs, Fermi and Kepler, mostly have some or all of the following deficiencies: higher latencies of microprocessors, less SMs, less CUDA cores per SM, smaller global memories, smaller maximum capacity for grid size and more power consumption while operating. A comparison will be given later to show how Maxwell GPUs are slightly better than Kepler GPUs and much faster than Fermi GPUs. All three architectures were used to show that even old GPUs provide better performance for computations in the current study over the most recent i7 CPUs without modification in the GPU codes.

7.1. [H] and [G] assembly comparison

The performance of the assembly of the two influence matrices $[H]$ and $[G]$ is combined for both the GPU and CPU codes. At 16 boundary elements for the rectangular plate both codes compute the coefficients in a few milliseconds (Fig. 8), however as the amount of elements on the boundary increases, the significance of parallelism is highlighted.

In a time of one second, one could possibly compute coefficients for a plate with 100 elements on the CPU, or carry out calculations for 1000 elements on the Kepler GPU (both single precision). As the limit of the Kepler GPU RAM is 2 GB, a plate with 5000 elements could be solved, where the GPU would be faster than the CPU code by 108 times at single precision accuracy, whereas the single precision GPU code is around 4.5 times faster than the double precision GPU code.

The GPU's performance is calculated using FLOPS. FLOPS is an acronym for Floating-point Operations Per Second, which is a benchmark measurement for assessing the speed of microprocessors while performing instructions. Floating-point operations include the number

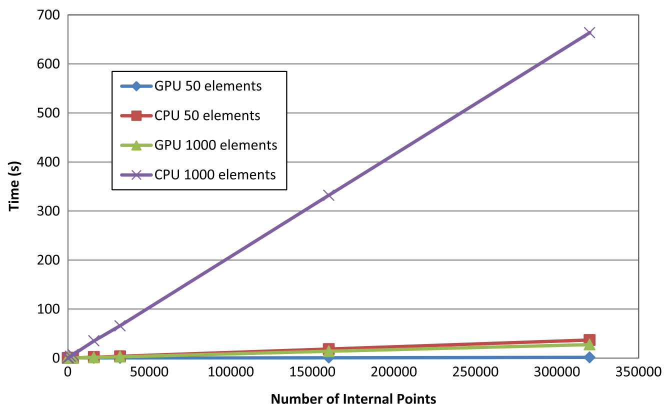

Fig. 10. A time comparison for internal point calculation between CPU code and GPU code.

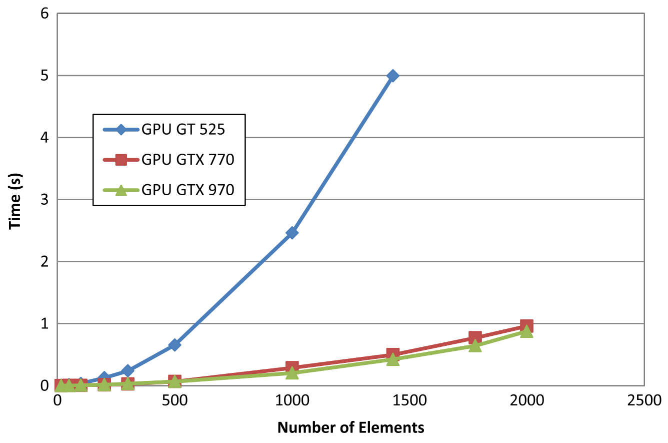

Fig. 11. Different GPU Architectures Performance on the same GPU code for influence matrices computation.

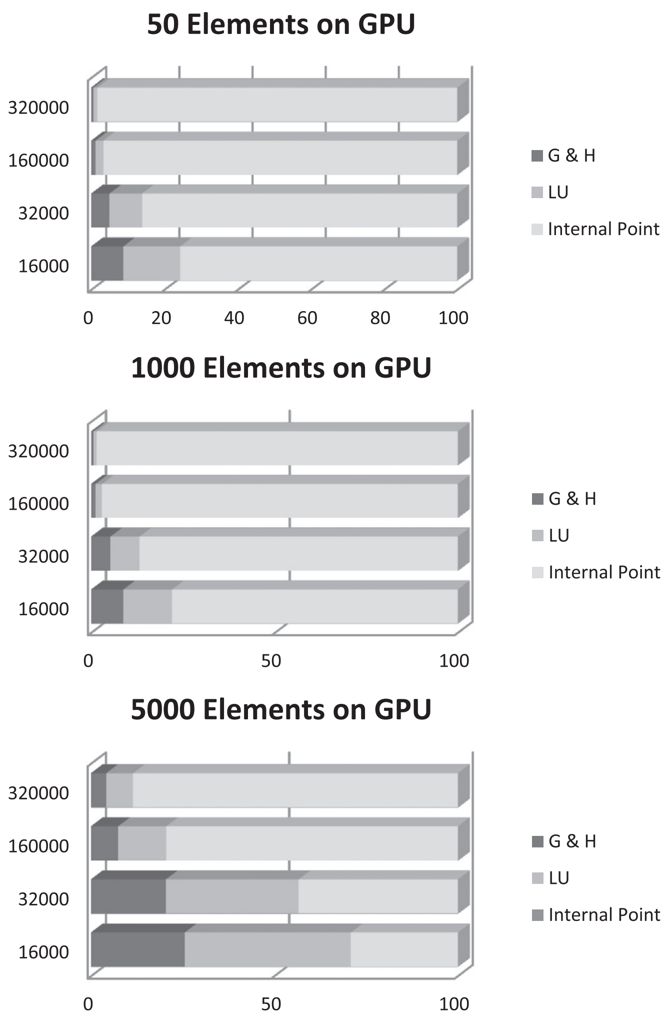

Fig. 12. The percentage of total time each of the three parts of the constant GPU code take to execute in parallel.

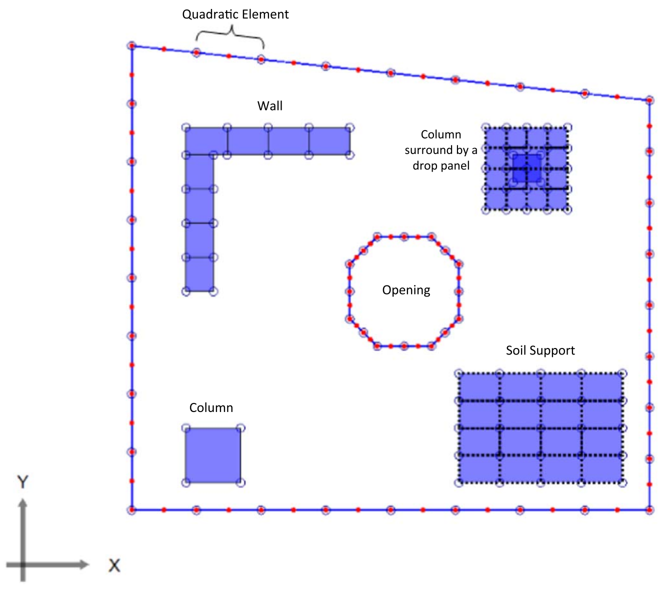

Fig. 13. A numerical model in the PLPAK software, representing the various loading elements and supports available on the boundary and the domain.

of mathematical operations with fractional numbers that a microprocessor could perform within one cycle. A microprocessor's speed is defined by the number of cycles it can perform in one second (a.k.a internal clock speed). The product of the internal clock speed and the number of operations performed in one cycle gives the theoretical performance of a single microprocessor in a GPU. Performance of 1.8 TFLOPS are reached by the single precision GPU code, whereas the limit reached by the double precision GPU code is around 80 GFLOPS on the Geforce GTX 770. The speed of 1.8 TFLOPS means 1.8×10^{12} floating-point operations computed each second, which mainly compromises of additions and multiplications.

7.2. System of equations comparison

Solving the linear system of equation using the CPU code has proved to be not time efficient by [22]. Applying the LU decomposition code developed by CULA has a great effect on compute time, as a plate with 2000 elements could be solved in under 0.5 s using the GPU and would take 650 s using the CPU with single precision accuracy. The [A] matrix size is 6000×6000, whereas the solution vector is 1×6000, proving that the $O(3N)^2$ complexity that would require $O(3N)^3$ operations to utilize the LU decomposition technique. Fig. 9 demonstrates that it is impractical to solve a plate with 5000 elements, which would take 2.5 h to complete using only the CPU whereas it would take around 4 s on the GPU.

7.3. Internal points comparison

The quantity of points inside the domain could vary independent of the number of boundary elements on the borders, yet higher quantities of internal points would represent a smoother demonstration of displacement contours. On the GPU, when the grid is initialized to compute the **UIPMdev** matrix, the grid size is defined in the y-direction by the number of boundary elements (m) and in the x-direction by the number of internal points (n). A relationship between m and n is demonstrated in Fig. 10, where m ranges from 50 to 1000

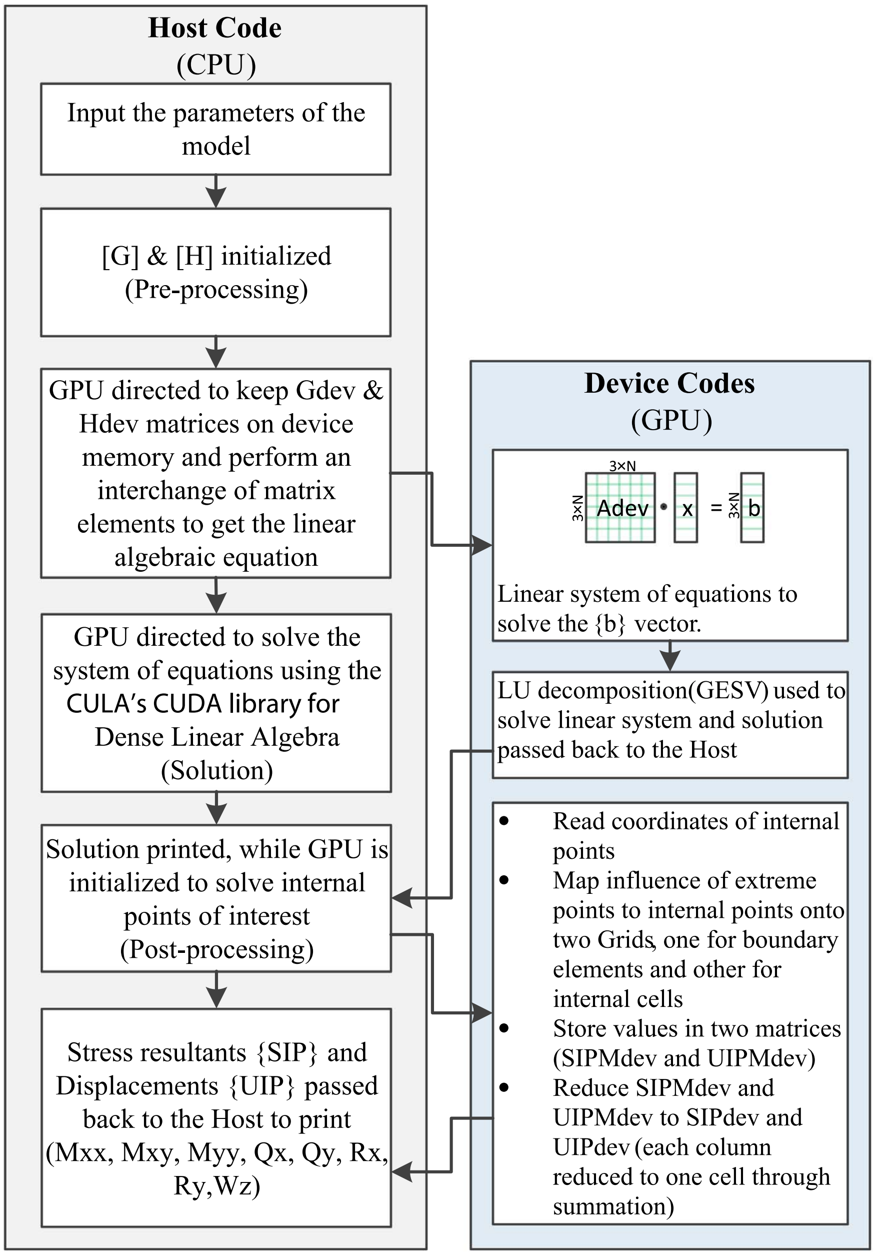
Host Code
(CPU)
Input the parameters of the model
[G] & [H] initialized
(Pre-processing)
GPU directed to keep Gdev & Hdev matrices on device memory and perform an interchange of matrix elements to get the linear algebraic equation
GPU directed to solve the system of equations using the CULA's CUDA library for Dense Linear Algebra
(Solution)
Solution printed, while GPU is initialized to solve internal points of interest
(Post-processing)
Stress resultants {SIP} and Displacements {UIP} passed back to the Host to print
(Mxx, Mxy, Myy, Qx, Qy, Rx, Ry,Wz)
Device Codes
(GPU)
3×N
3×N
Adev • x = b
3×N
Linear system of equations to solve the {b} vector.
LU decomposition(GESV) used to solve linear system and solution passed back to the Host
Read coordinates of internal points
Map influence of extreme points to internal points onto two Grids, one for boundary elements and other for internal cells
Store values in two matrices (SIPMdev and UIPMdev)
Reduce SIPMdev and UIPMdev to SIPdev and UIPdev (each column reduced to one cell through summation)

Fig. 14. The flow of data between the host and the device in the new quadratic GPU code.

elements and n ranges from 50 to 320,000 points. The figure demonstrates the comparison between serial-computed against parallel computed internal points, where for large numbers of m and n the GPU code is faster by around 25 times. At m=1000 and n=320,000 the CPU code took 660 s, whereas the GPU code took 27.5 s

7.4. Maxwell vs. Fermi vs. Kepler

The GPU code is rerun on a Fermi architecture to observe code performance against different NVIDIA architectures. The implementation on the GPU is developed to be compatible with all architectures. An old GPU, Geforce GT 525 M (Fermi), is compared to two newer GPUs, Geforce GTX 770 (Kepler) and GTX 970 (Maxwell), whereas hardware comparisons are shown in Table 1. The Fermi GPU has 1 GB of global memory, thus the maximum number of boundary elements is 1500 elements, which would require around 800 MB of memory. At the maximum number of elements of the Fermi GPU, the Kepler GPU outperforms by 10 times whereas the Maxwell GPU performs around even 10% better than the Kepler GPU, as demonstrated in Fig. 11. The GPUs' architecture differences provide an explanation for their performances with the constant GPU code.

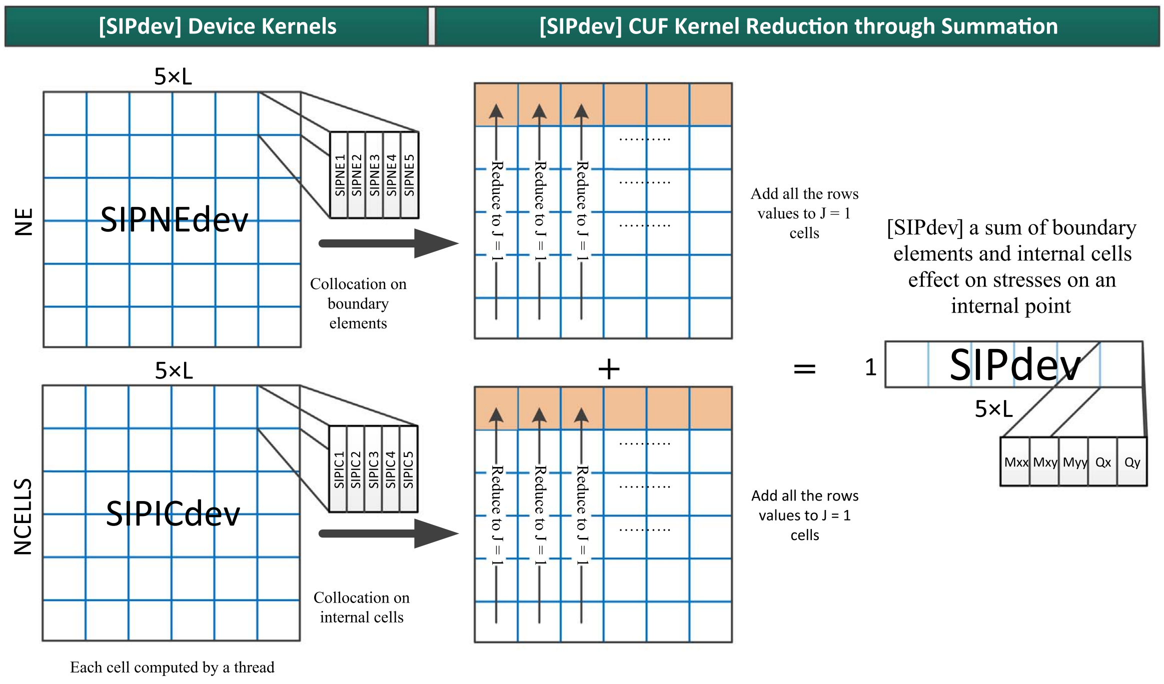
[SIPdev] Device Kernels
[SIPdev] CUF Kernel Reduction through Summation
5×L
NE
SIPNEdev
SIPNE1
SIPNE2
SIPNE3
SIPNE4
SIPNE5
Collocation on boundary elements
Reduce to J = 1
Add all the rows values to J = 1 cells
[SIPdev] a sum of boundary elements and internal cells effect on stresses on an internal point
+
=
1
SIPdev
5×L
Mxx
Mxy
Myy
Qx
Qy
NCELLS
SIPICdev
SIPIC1
SIPIC2
SIPIC3
SIPIC4
SIPIC5
Collocation on internal cells
Each cell computed by a thread

Fig. 15. The flow of **SIPdev** computation on the GPU through two device kernels and then reduction through summation.

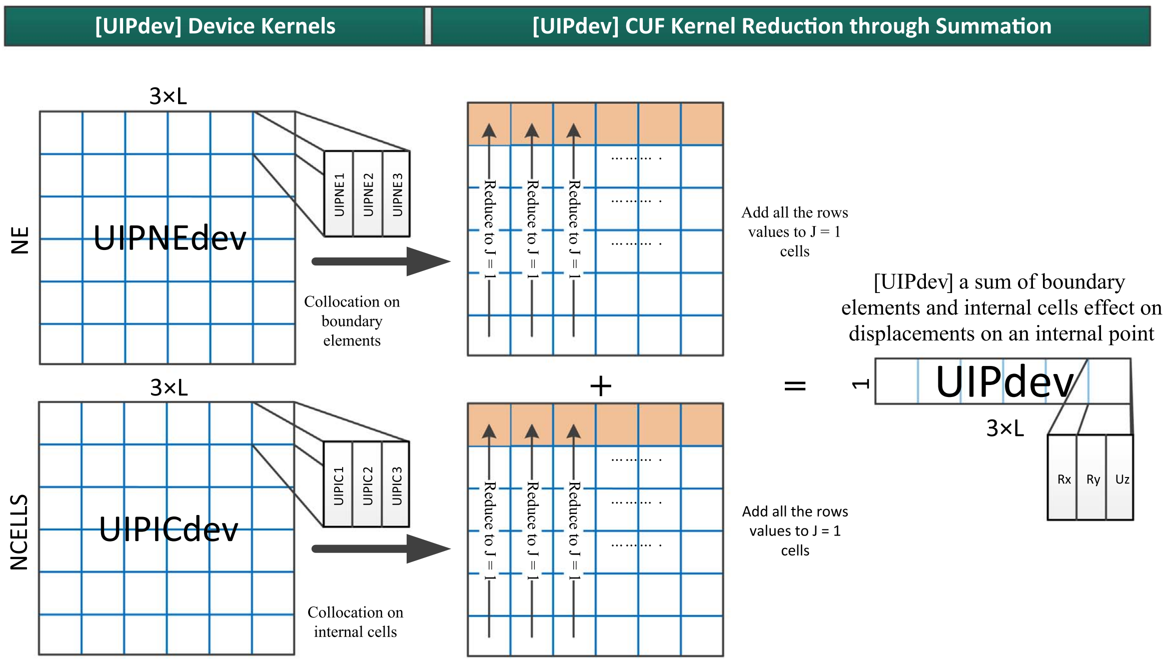
[UIPdev] Device Kernels
[UIPdev] CUF Kernel Reduction through Summation
3×L
NE
UIPNEdev
UIPNE1
UIPNE2
UIPNE3
Collocation on boundary elements
Reduce to J = 1
Add all the rows values to J = 1 cells
[UIPdev] a sum of boundary elements and internal cells effect on displacements on an internal point
+
=
1
UIPdev
3×L
Rx
Ry
Uz
NCELLS
UIPICdev
UIPIC1
UIPIC2
UIPIC3
Collocation on internal cells

Fig. 16. The flow of **UIPdev** computation on the GPU through two device kernels and then reduction through summation.

7.5. Evaluation of results

As the accuracy remained consistent between the two codes, the time comparison performed on the three distinct parts of each code is re-evaluated to identify exactly which parts of the code spend the most absolute time. The numerical example of constant boundary elements made clear that at double precision accuracy: influence matrices assembly is reduced about 25 times, system solution is reduced about 1300 times, and internal point calculation is reduced about 25 times. This implies that the first and third parts of the code have the same relative speed-ups. Further re-evaluation in Fig. 12 demonstrates three different models for the same numerical model discussed above, with three discretizations: 50, 1000 and 5000 boundary elements. In each case the whole GPU codes were executed, whereas each case is performed four times with varying internal points count from 16,000 points to 320,000 points. When the absolute time is taken into account, internal points exceeding 100,000 are computed in a much longer time than the other two parts, and takes around 70–98% of the total computation time.

It is evident that the internal point calculation reprogramming benefited most from the GPU's parallelism, whereas the solution of the system of equations and influence matrices assembly came in second and third respectively. The reason for the second part of the code taking longer computation time is due to the O(N^3) complexity, whereas the first part is of O(3N)^2 complexity. It should be noted that boundary solution accuracy difference is negligible between 5000 and 1000 elements cases, therefore increasing the number of elements drastically will not necessarily attain better solutions and using less elements is favorable to attain faster results. Also in practical plate bending problems, a huge number of boundary elements are not required (about 100–200 elements for huge applications), whereas internal results contours are more representative of the model's analysis. Therefore, as will be demonstrated later for quadratic elements, it is vital to parallelize the solution of linear system of equations and the solution of internal points for the implementation of practical shear-deformable plates that use quadratic elements.

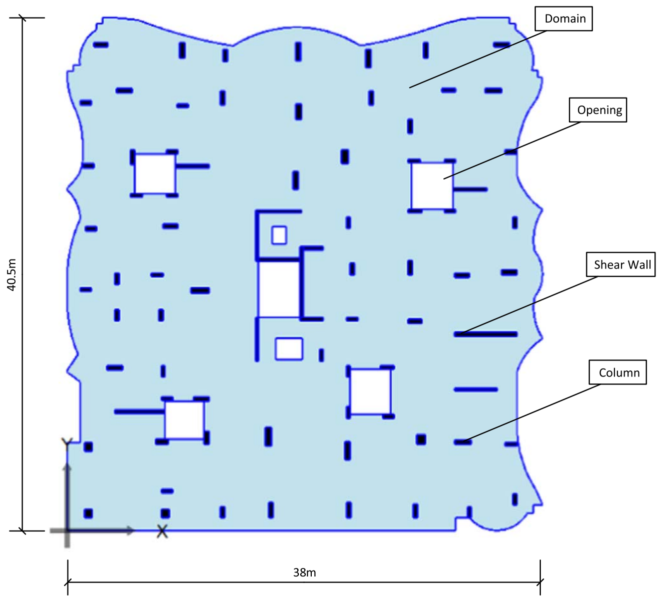

Fig. 17. Case 1, Irregular building slab with columns and walls.

8. Quadratic elements

In this section, the proposed GPU approach is implemented using quadratic elements.

8.1. Plate bending package (PLPAK)

A developed serial software, the PLPAK [26], is being developed to accurately perform structural analysis of building slabs and foundations using the boundary element method. Incorporated in the software are the BEM for plate bending using quadratic elements and a GUI integrating several engineering tools. In this section GPU computing is implemented on the PLPAK software to parallelize all necessary routines.

To model real floors with precise geometry and under realistic gravitational-direction loads, several other internal "supports" were added to the plate bending formulation in Section 2. and coded into the software. The internal additions are identified as "Internal Supporting Cells" in the formulation and include: Columns, Walls, Beams, Soil supports, Drop panels and any other supporting element in the form of stiffness matrix as in Fig. 13.

The PLPAK processes data with algorithms similar to those of the constant element code discussed earlier in addition to taking internal cells into consideration. The serial code for the PLPAK goes through three different steps: (a) computation of influence matrices coefficients, (b) solution of linear system of equations, and (c) internal point computation. As discussed in the Section 7.5, it is recommended to modify only the later two parts of the code for efficient parallel processing. The influence matrices include elements' and internal cells' influences, a single device kernel that could do both would be too hefty to write and to no avail as the BEM converges to good results at low numbers of elements.

8.2. Exporting relevant matrices from the PLPAK

The PLPAK software is capable of exporting data at certain intervals of its processes, of which is the extraction of the $[A]$ matrix, the $\{b\}$

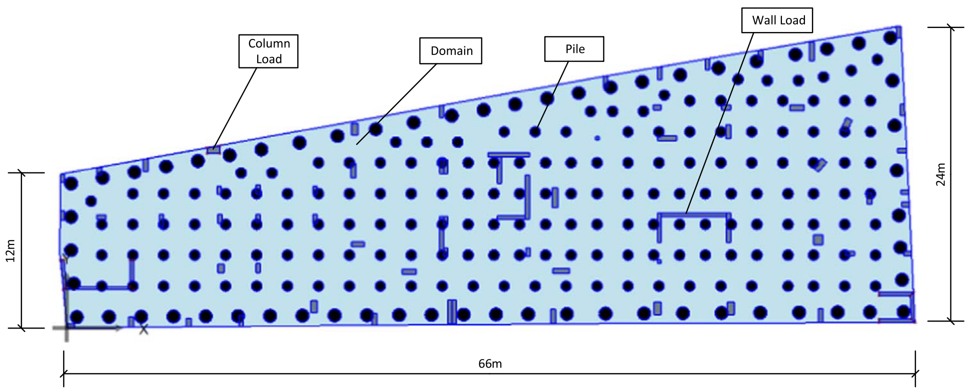

Fig. 18. Case 2, Raft foundation on piles with column and wall loads.

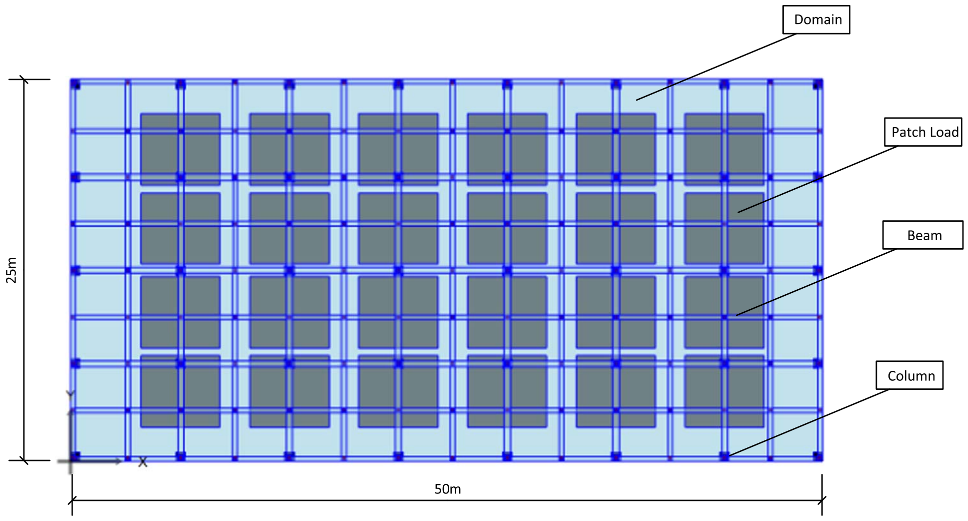

Fig. 19. Case 3, Building slab with a multitude of beams, columns and drops.

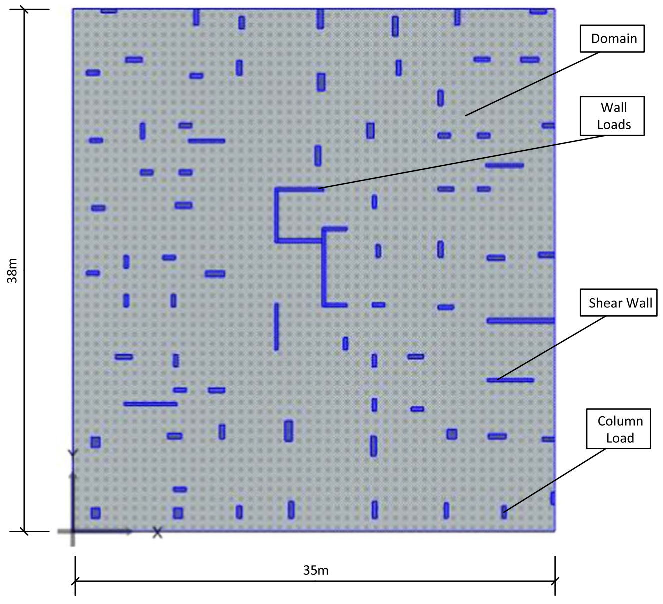

Fig. 20. Case 4, Building raft with a multitude of soil supports and columns loads.

Table 2
Properties and loading of case studies.

Case Number	Number of Boundary Elements	Number of Internal Supports	Dimensions	Material Properties	Type of Loads
1	200, 600, 1800, 2600	70 Columns +Walls	38 m×40.5 m	Thickness: 0.2 m Plate's E: 2,410,000 t/m^2	uniform load +own weight
2	50, 200, 1000, 2000	63 Piles	66 m×24 m	Thickness: 1.5 m Plate's E: 2,210,000 t/m^2	uniform load +own weight +walls, columns
3	50, 150, 400, 1000	40 Columns, 2000 beam divisions	50 m×25 m	Thickness: 0.25 m Plate's E: 2,500,000 t/m^2	uniform load +own weight
4	16	Soil Supports range: 72, 332, 728, 1330, 3031	35 m×38 m	Thickness: 1.4 m Plate's E: 2,200,000 t/m^2	81 column +wall loads

vector, the boundary tractions and displacements vectors, and the internal points results vectors. The matrices could be exported to be further processed by an external source. These matrices are made use of in this GPU implementation to both incorporate them in GPU external codes and to check on results processed on the GPU. This would leave the only part of the process that is not parallelized to be the formation of the $[A]$ matrix and the $[b]$ vector, which the original PLPAK software will always serially compute.

8.3. The proposed PLPAK parallel approach

The PLPAK's serial code is modified on CUDA Fortran targeting the solution of equations and internal points calculations parts. The solution of equations is done using the LU Decomposition to attain tractions and displacements on boundary nodes, similar to the approach of the constant element, whereas the internal point computation is processed in a different way discussed later. In Fig. 14 a flow chart shows the current flow and handling of data on the Host and the Device.

Each internal point has five stress resultants and three generalized displacements that need to be calculated. These values are computed through two distinct device kernels in the following order: (a) a stress resultants kernel for internal points due to the influences of all elements and cells storing the values in the device array **SIPdev**, (b) a generalized displacements kernel for internal points due to the influences of all elements and cells storing the values in the device array **UIPdev**. In both device kernels two matrices are primarily initiated: one of each element's influence on each point and one of each cell's influence on each point, followed by a reduction summation for both matrices to attain a one dimensional array of the results. This process is demonstrated in Figs. 15 and 16, and targets the spread out of the entirety of all the collocations on the GPU's grid to minimize any serial work done by a thread, with the exception of Gauss points calculations as only a mere 10 Gauss points were chosen.

Where L is the number of internal points, NE is the number of boundary elements, NCELLS is the number of internal cells, Mxx is the moment in x-direction, Mxy is the torsional moment, Myy is the

Table 3
Linear system of equations solution time for all case studies.

Case Info	**Case 1**								Maximum Speed Up
Number of Elements	200		600		1800		2600		**661 X**
Hardware	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU	
Computation Time (s)	2.774	0.063	182.066	0.6089	6300	10.4	**15531**	**23.493**	
Case Info	**Case 2**								Maximum Speed Up
Number of Elements	50		200		1000		2000		**606 X**
Hardware	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU	
Computation Time (s)	0.805	0.031	18.249	0.152	1455	4.167	**9930**	**16.366**	
Case Info	**Case 3**								Maximum Speed Up
Number of Elements[a]	50		150		400		1000		**706 X**
Hardware	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU	
Computation Time (s)	873.7	2.731	1107.357	3.6619	2169	4.161	**8054**	**11.402**	
Case Info	**Case 4**								Maximum Speed Up
Number of Soil Supports[b]	332		728		1330		3031		**579 X**
Hardware	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU	
Computation Time (s)	1.081	0.04	22.127	0.174	218.2	0.683	**3235**	**5.59**	

[a] Number of beam divisions constant at 2000 elements.
[b] Number of boundary elements constant at 16 elements.

moment in y-direction, Qx is the shear in x-direction, Qy is the shear in y-direction, Rx is the rotation in x-direction, Ry is the rotation in y-direction, and Uz is the displacement in z-direction.

9. Numerical results for quadratic elements

Four different practical building floors with distinct functions were analyzed using the serial and parallel PLPAK codes. The speed of analysis of the practical building floors were recorded in the cases of serial and parallel computations, and then compared. The building floors are based on the formulations in Rashed et al. [16–19], and they are:

- Case 1: Irregular building slab with columns and walls (Fig. 17).
- Case 2: Raft foundation on piles with column and wall loads (Fig. 18).
- Case 3: Building slab with a multitude of beams, columns and drops (Fig. 19).
- Case 4: Building raft with a multitude of soil supports and columns loads (Fig. 20).

The geometry, properties and loading of each case is stated in Table 2. The Geforce GTX 970 GPU is employed.

9.1. Results of solution of system of equations

The four Cases' boundary solutions are solved on both the CPU and the GTX 970 GPU using LU Decomposition and the results can be seen in Table 3. Cases 1, 2, and 3 had a constant number of internal supports whereas the number of quadratic elements were increased. In Case 4 the number of boundary elements are invariable, whereas the number of soil supports where increased. As the number of boundary elements and internal supports were increased in all case studies, the CPU's time rose exponentially reaching a few hours in some cases whereas the GPU processing time remained as a few seconds.

Increasing the number of boundary elements is computationally different to increasing the number of internal supports inside any given model. Both intensify the overall run time of both the GPU and CPU processing time but there is an important comparison that needs to be made. The difference is due to $O(9N)^2$ complexity of the quadratic boundary element collocation against the $O(3N)^2$ complexity of an internal support's collocation.

In case 1, 2 and 3 the number of internal supports (columns, walls, beams and piles) are well defined and could not be adjustable due to the parameters of the model, thus the number of quadratic boundary elements for each case was the only quantity that was increased and its computation time was measured. For case 4 the number of quadratic boundary elements was optimized at 16 elements and soil supports were increased from 332 to 3031. Soil supports were increased to more precisely model the different properties of the underlying soil by dividing the soil patches into smaller areas. While case 1 took 23.493 s to be computed on the GPU with 2600 quadratic elements, case 4 took 5.59 s to be computed with 3031 soil supports. In all cases the GPU's performance exceeded that

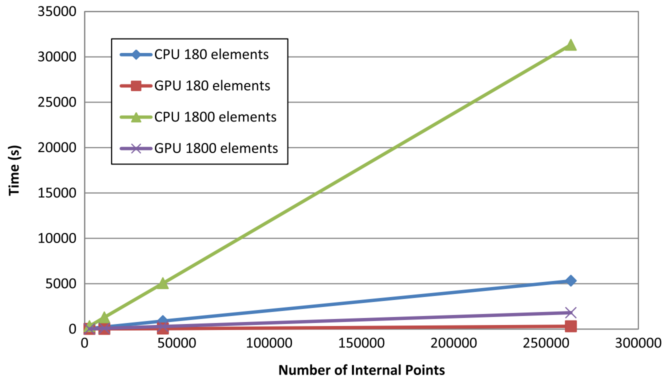

Fig. 21. A time comparison of internal point calculation for Case Study 1 between the CPU code and GPU code (double precision accuracy).

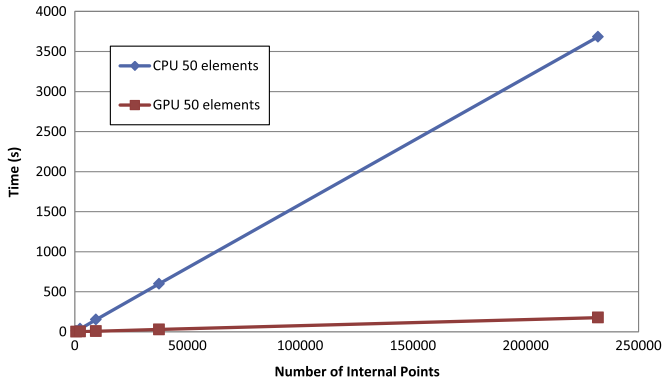

Fig. 22. A time comparison of internal point calculation for Case Study 2 between the CPU code and GPU code (double precision accuracy).

of the CPU at least to 579 times. This is similar to the solution of system of equations comparison made in Section 7.2.

9.2. Results for internal point computation

For internal point computation time comparison, Case 1 and 2 were considered. Internal points were evenly distributed throughout the perimeter of the models at spacing ranging from 2 m to 0.02 m. In both cases a low number of elements is compared to a large number of elements placed on the boundary. Figs. 21 and 22 demonstrates the drastic increase in time caused by the vast increase in internal points in both cases studied. In Case 1 specifically, around quarter a million points could be computed in around 9 h on the CPU, whereas with the same accuracy the same results could be reached in 30 min on the GPU using the proposed PLPAK parallel approach. A very fine mesh for domain results contours could be produced in much less time than was previously possible.

10. Conclusions

In this paper, a direct boundary element method for shear-deformable plate bending is accelerated by implementation on a graphics processing unit to condense the calculation time through parallel computing. Two constant BEM codes are created, one that performs calculations on the CPU and the other on the GPU. With accuracy remaining consistent between the two codes, a time comparison is performed on the three distinct parts of each code: influence matrix assembly, linear algebraic system solution and internal point calculation. Taking into account GPU memory limit, a numerical example with a great number of boundary elements showed that influence matrices assembly could be reduced 25 times, system solution reduced 1300 times, and internal point calculation reduced 25 times. The proposed GPU code is then executed on an older NVIDIA GPU without any modifications and the influence matrices assembly time is recorded. Although both GPU runs exceed CPU performance, the newer GPU processed data 10 times faster at the largest possible number of elements. The proposed code is capable of running even faster on more modern hardware in the future, whereas the CPU code would still be impractical as it consumes excessive time for large models.

As a sub-conclusion to the constant BEM GPU results, an evaluation is applied on the proposal of a GPU enhanced PLPAK in order to compute actual building floors using the quadratic BEM. Two worthwhile processes were parallelized, which were the solution of linear system of equations and the computation of internal points. The computation and assembly of the influence coefficients remains serial as boundary elements do not need to be massively discretized for building floors. A GPU implementation of the PLPAK is presented and tested on four actual floors, including a raft and piled-raft foundation. The GPU implementation proved that very large models could be solved in mere minutes whereas the CPU codes solved these problems in several hours.

The last result opens the door for other issues concerning plate bending which needs to be addressed. These issues include the parallel implementation of multi-storey building analysis, time history analysis and composite material integration; currently considered by the authors.

Acknowledgements

This project was supported financially by the Science and Technology Development Fund (STDF), Egypt, Grant No 14910.

The authors would like to acknowledge the support of Prof Amr Amin Adly, vice president of Cairo University for his support.

Appendix A. The fundamental solution kernels in Section 2

The expression for the kernel U^*_{ijk} is given by [20]:

$$U^*_{\alpha\beta\gamma}=\frac{1}{4\pi r}[(4A+2zK_1+1-\nu)\times(\delta_{\beta\gamma}r_\alpha+\delta_{\alpha\gamma}r_{,\beta})-2(8A+2zK_1+1-\nu)r_{,\alpha}r_{,\beta}r_{,\gamma}+(4A+1+\nu)\delta_{\alpha\beta}r_{,\gamma}] \tag{A.1}$$

$$U^*_{\alpha\beta3}=\frac{-(1-\nu)}{8\pi}\times[(2\frac{(1+\nu)}{(1-\nu)}\ln z-1)\delta_{\alpha\beta}+2r_{,\alpha}r_{,\beta}] \tag{A.2}$$

$$U^*_{3\beta\gamma}=\frac{\lambda^2}{2\pi}[B\delta_{\gamma\beta}-Ar_{,\gamma}r_{,\beta}] \tag{A.3}$$

$$U^*_{3\beta3}=\frac{1}{2\pi r}r_{,\beta} \tag{A.4}$$

The expression for the kernel T^*_{ijk} is given by [20]:

$$P^*_{\alpha\beta\gamma}=\frac{D(1-\nu)}{4\pi r^2}\{(4A+2zK_1+1-\nu)\times(\delta_{\gamma\alpha}n_\beta+\delta_{\gamma\beta}n_\alpha)+(4A+1+3\nu)\delta_{\alpha\beta}n_\gamma-(16A+6zK_1+z^2K_0+2-2\nu)\times[(n_\alpha r_{,\beta}+n_\beta r_{,\alpha})r_{,\gamma}+(\delta_{\gamma\alpha}r_{,\beta}+\delta_{\gamma\beta}r_{,\alpha})r_{,n}]- \\ 2(8A+2zK_1+1-\nu)\times(\delta_{\alpha\beta}r_{,\gamma}r_{,n}+n_\gamma r_{,\alpha}r_{,\beta})+4(24A+8zK_1+z^2K_0+2-2\nu)\times r_{,\alpha}r_{,\beta}r_{,\gamma}r_{,n}\} \tag{A.5}$$

$$P^*_{\alpha\beta3}=\frac{D(1-\nu)\lambda^2}{4\pi r^2}[(2A+zK_1)(r_{,\beta}n_\alpha+r_{,\alpha}n_\beta)-(2A+zK_1)r_{,\alpha}r_{,\beta}r_{,n}+2A\delta_{\alpha\beta}r_{,n}] \tag{A.6}$$

$$P^*_{3\beta\gamma}=\frac{D(1-\nu)\lambda^2}{4\pi r^2}[(2A+zK_1)(\delta_{\gamma\beta}r_{,n}+r_{,\gamma}n_\beta)+2An_\gamma r_{,\beta}-2(4A+zK_1)r_{,\gamma}r_{,\beta}r_{,n}] \tag{A.7}$$

$$P^*_{3\beta3}=\frac{D(1-\nu)\lambda^2}{4\pi r^2}[(z^2B+1)n_\beta-(z^2A+2)r_{,\beta}r_{,\alpha}] \tag{A.8}$$

where:

$$r^2=(x_\alpha(\mathrm{x})-x_\alpha(\xi))(x_\alpha(\mathrm{x})-x_\alpha(\xi)) \tag{A.9}$$

$$z=\lambda r \tag{A.10}$$

$$r_{,\alpha}=\frac{\partial r}{\partial x_\alpha(x)} \tag{A.11}$$

$$A=A(z)=K_0(z)+\frac{2}{z}[K_1(z)-\frac{1}{z}] \tag{A.12}$$

$$B=B(z)=K_0(z)+\frac{1}{z}[K_1(z)-\frac{1}{z}] \tag{A.13}$$

in which K_0 and K_1 are modified Bessel functions.

References

[1] Takahashi T, Hamada T. GPU-accelerated boundary element method for Helmholtz' equation in three dimensions. Int J Numer Methods Eng 2009;80(10):1295–321.

[2] Komatitsch D, Michéa D, Erlebacher G. Porting a high-order finite-element earthquake modeling application to NVIDIA graphics cards using CUDA. J Parallel Distrib Comput 2009;69:451–60.

[3] Yokota R, Barba, RA.. Chapter 9-Treecode and Fast Multipole Methodfor N-Body Simulation with CUDA. GPU Computing Gems Emerald Edition, A volume in Applications of GPU Computing Series; 2011 pp. 113–132

[4] Hadi MF, Esmaeili SA. CUDA Fortran acceleration for the finite-difference time-domain method. Comput Phys Commun 2013;184:1395–400.

[5] NVIDIA, CUDA (Compute Unified Device Architecture) Programming Guide, Version 7.0, NVIDIA Corporation, Santa Clara, CA, USA, 240 pages; (March 2015).

[6] Tsuchiyama R, Nakamura T, Iizuka T, Asahara A. The open CL programming book. Tokyo: Fixstars Corporation; 2010. p. 324.

[7] Labaki J, Ferreira LOS, Mesquita E. Constant Boundary Elements on graphics hardware: a GPU-CPU complementary implementation [Rio de Janeiro]. J Braz Soc Mech Sci Eng 2011;33(4).

[8] Wang Y, Wang Q, Wang G, Huang Y, Wei Y, Boundary element parallel computation for 3D elastostatics using CUDA. In: Proceedings of the ASME 2011 international design engineering technical conferences &, computers and information in engineering conference, IDETC/CIE 2011, 29–31, (August 2011), Washington, DC, USA, DETC2011-47981.

[9] Iuspa L, Fusco P, Ruocco E. An improved GPU-oriented algorithm for elastostatic analysis with boundary element method. Comput Struct 2015;146:105–16.

[10] Coopera CD, Bardhanb JP, Barbaa LA. A biomolecular electrostatics solver using Python, GPUs and boundary elements that can handle solvent-filled cavities and Stern layers. Comput Phys Commun 2014;185(3):720–9.

[11] Wanga Y, Wangb Q, Deng X, Xiab Z, Yana J, Hua Xuc . Graphics processing unit (GPU) accelerated fast multipole BEM with level-skip M2L for 3D elasticity problems. Adv Eng Softw 2015;82:105–18.

[12] Li Y, Zhang J, Yu L, Lu C, Li C. GPU-accelerated regular integrationand singular integration in boundary face method. Aust J Mech Eng 2015;13(3):163–71.

[13] Reissner E. On the theory of bending of elastic plates. J Math Phys 1944;23:184–91.

[14] Berrebia CA, Telles JCF, Wrobel LC. Boundary element techniques: theory and applications in engineering. Berlin, Heidelberg: Springer; 1984.

[15] Vander Weeën F. Application of the boundary integral equation method to Reissner's plate model. Int J Numer Methods Eng 1982;18:1–10.

[16] Rashed YF. Boundary element modelling of flat plate floors under vertical loading. Int J Numer Methods Eng 2005;62:1606–35.

[17] Rashed YF. A boundary/domain element method for analysis of building raft foundations. Eng Anal Bound Elem 2005;29:859–77.

[18] Mehanny SSF, Mehanny SSF, Rashed YF. A probabilistic boundary element method applied to the pile dislocation problem. Eng Struct 2011;33:2919–30.

[19] Shaaban AM, Rashed YF. A coupled BEM-stiffness matrix approach for analysis of shear deformable plates on elastic half space. Eng Anal Bound Elem 2013;37:699–707.

[20] Rashed YF, AIiabadi MH, Brebbia CA. On the evaluation of the stresses in the BEM for Reissner plate-bending problems. Appl Math Model 1997;21:155–63.

[21] ⟨http://www.geforce.com/hardware⟩.

[22] ⟨http://www.pgroup.com/resources/cudafortran.htm⟩.

[23] ⟨http://www.nvidia.com/object/cuda_home_new.html⟩.

[24] Humphrey JR, Price DK, Spagnoli KE, Paolini AL, Kelmelis EJ. CULA: Hybrid GPU Accelerated Linear Algebra Routines, SPIE Defense and Security Symposium (DSS); 2010.

[25] ⟨https://www.pgroup.com/lit/articles/insider/v2n3a1.htm⟩.

[26] Rashed YF, Mobasher EM. Products & practice spotlight: a new tool for structural designers. Concr Int 2012;34(10).