18TH WORLD CONFERENCE ON EARTHQUAKE ENGINEERING
WCEE2024
MILAN

MILAN, ITALY
30TH JUNE - 5TH JULY 2024
www.wcee2024.it

ACCELERATING FINITE-ELEMENT STRUCTURAL ELASTIC DYNAMIC ANALYSIS USING GPU COMPUTING

G.A. Araújo R.[1], B.G. Simpson[2], M. Zhu[3] & M.H. Scott[4]

[1] PhD student, Department of Civil and Environmental Engineering, Stanford University, Stanford, CA, United States, garaujor@stanford.edu

[2] Assistant Professor, Department of Civil and Environmental Engineering, Stanford University, Stanford, CA, United States

[3] Research Associate, School of Civil and Construction Engineering, Oregon State University, Corvallis, OR, United States

[4] Professor, School of Civil and Construction Engineering, Oregon State University, Corvallis, OR, United States

Abstract: *The demand for high-performance computing resources has led to a paradigm shift towards massive parallelism using graphics processing units (GPUs) in many scientific disciplines, including machine learning, robotics, quantum chemistry, molecular dynamics, and computational fluid dynamics. In earthquake engineering, artificial intelligence and data-driven methods have gained increasing attention for leveraging GPU-computing for seismic analysis and evaluation for structures and regions. However, in finite-element analysis (FEA) applications for civil structures, the progress in GPU-accelerated simulations has been slower due to the unique challenges of porting structural dynamic analysis to the GPU, including the reliance on different element formulations, nonlinearities, coupled equations of motion, implicit integration schemes, and direct solvers. This research discusses these challenges and potential solutions to fully accelerate the dynamic analysis of civil structural problems. To demonstrate the feasibility of a fully GPU-accelerated FEA framework, a pilot GPU-based program was built for linear-elastic dynamic analyses. In the proposed implementation, the assembly, solver, and response update tasks of FEA were ported to the GPU, while the central-processing unit (CPU) instructed the GPU on how to perform the corresponding computations and off-loaded the simulated response upon completion of the analysis. Since GPU computing is massively parallel, the GPU platform can operate simultaneously on each node and element in the model at once. As a result, finer mesh discretization in FEA will not significantly increase run time on the GPU for the assembly and response update stages. Work remains to refine the program for nonlinear dynamic analysis.*

1 Introduction

Finite-element analysis for structural dynamics has widespread use in academia and industry for the seismic design and performance assessment of civil structures (TBI, 2017; Wilson, 2010). However, the computational cost associated with these simulations often restricts the practical level of resolution used in the numerical model, which is defined by both model size and mesh refinement. For instance, executing a single response history analysis for a tall building using a realistic three-dimensional model can take several hours or even days (Lu and Guan, 2017). Moreover, complex simulations involving soil-structure (Elgamal *et al.*, 2008;

McCallen *et al.*, 2022) and fluid-structure interactions (Gimenez *et al.*, 2017; Motley *et al.*, 2016; Zhu and Scott, 2014) typically demand significant computing resources and extended run times (Simpson *et al.*, 2023). The need for high-performance computing (HPC) resources also increases for parametric problems such as uncertainty propagation or optimization, which require many realizations. Assessing ground motion variability at the regional-level (McCallen *et al.*, 2022) also often demands simplified or reduced-order numerical models to reduce computational cost.

Parallel processing on central processing units (CPUs) has long been employed to accelerate finite-element analyses (FEA) by partitioning the spatial domain across multiple processors (Chiang and Fulton, 1990; Mackerle, 1996, 2003, 2004; McKenna and Fenves, 2000; McKenna, 1997; Topping and Khan, 1996). However, this approach encounters limitations, because the split subdomains remain coupled, necessitating communication between parts of the analysis running on different cores. As more cores are utilized, the communication overhead can cause performance plateaus, where adding extra cores does not result in decreased execution time (Amdahl, 1967; Jeremić and Jie, 2008).

In contrast, graphics processing units (GPUs) are characterized for their massive parallel computing capabilities, employing thousands of cores that can run operations on large volumes of data at a time. While research in seismic wave propagation (Komatitsch *et al.*, 2009; O'Reilly *et al.*, 2022; Roten *et al.*, 2016) and biomechanics (Johnsen *et al.*, 2015; Joldes *et al.*, 2010; Mafi and Sirouspour, 2014; Taylor *et al.*, 2008) have successfully employed massively parallel GPU computing for FEA and finite differences, progress for the structural dynamic analysis of civil structures has been comparatively slow. Challenges include diverse element formulations, material along with geometric nonlinearities, coupled equations of motion, implicit integration schemes, and reliance on direct solvers that have been conceived to execute in parallel on CPUs.

This paper explores the potential of using GPUs to accelerate the dynamic analysis of civil infrastructure using finite-element analysis. To begin with a simple test case, a pilot program was built in CUDA to fully GPU-accelerate a linear-elastic dynamic analysis. In contrast to previous partially-accelerated GPU implementations of FEA (Bartezzaghi *et al.*, 2015; Georgescu *et al.*, 2013; Kang *et al.*, 2014; Posey and Wang, 2012), the platform developed in this study ported all major FEA tasks to the GPU, including the matrix assembly, equation solver, and domain update. The CPU, acting as the host, instructed the GPU, functioning as the device, on how to perform the computations (NVIDIA, 2017a), but the output was only returned to the CPU upon completion of an analysis step. Future work will explore incorporating nonlinear behavior and the use of different element formulations and time-integration methods.

2 GPU implementation

To benchmark potential speedups, a simple platform was built to fully accelerate a linear-elastic dynamic analyses for frame structures (Simpson *et al.*, 2023). The source code was implemented in CUDA® C++ v.9.0, a parallel programming model developed by NVIDIA® for general computing on GPUs (NVIDIA, 2017a). The major components of the platform are illustrated in **Figure 1**. The following sections use the minimal example in **Figure 2** to illustrate the implementation in CUDA.

The program uses FEA to solve the equations of motion for an elastic multiple-degree-of-freedom (DOF) system:

$$\mathbf{M}\,\ddot{\mathbf{U}}_f + \mathbf{C}\,\dot{\mathbf{U}}_f + \mathbf{K}\,\mathbf{U}_f = \mathbf{P}_f(t) - \boldsymbol{P}_0(t), \tag{1}$$

where $\mathbf{M}$ = the system mass matrix $[N_f \times N_f]$; $\mathbf{C}$ = the system damping matrix $[N_f \times N_f]$; $\mathbf{K}$ = the elastic stiffness matrix $[N_f \times N_f]$; $\mathbf{U}_f$, $\dot{\mathbf{U}}_f$, and $\ddot{\mathbf{U}}_f$ = the free-DOF displacements, velocities, and accelerations $[N_f \times 1]$; $\mathbf{P}_f(t)$ = the free-DOF external dynamic loads, taken here to be only at the DOFs $[N_f \times 1]$; $\mathbf{P}_0(t)$ = the free-DOF assembled element loads $[N_f \times 1]$; and N_f = the number of free DOFs.

A time integration scheme is usually applied to a linear program representing a solution for the next time step based on **Eq. (1)**. In this study, the second-order differential equations of motion governing structural dynamics from **Eq. (1)** are discretized in time and solved using the unconditionally stable, implicit Newmark-beta time-stepping integration scheme. This method requires the solver to compute the solution at each time step n to a system of equations of the form:

$$\mathbf{A}\,\mathbf{x} = \mathbf{b}. \tag{2}$$

where $\mathbf{A}$ = the left-hand side, effective tangent stiffness matrix $[N_f \times N_f]$; $\mathbf{b}$ = the right-hand side, residual load vector $[N_f \times 1]$, and $\mathbf{x}$ = the solution vector $[N_f \times 1]$.

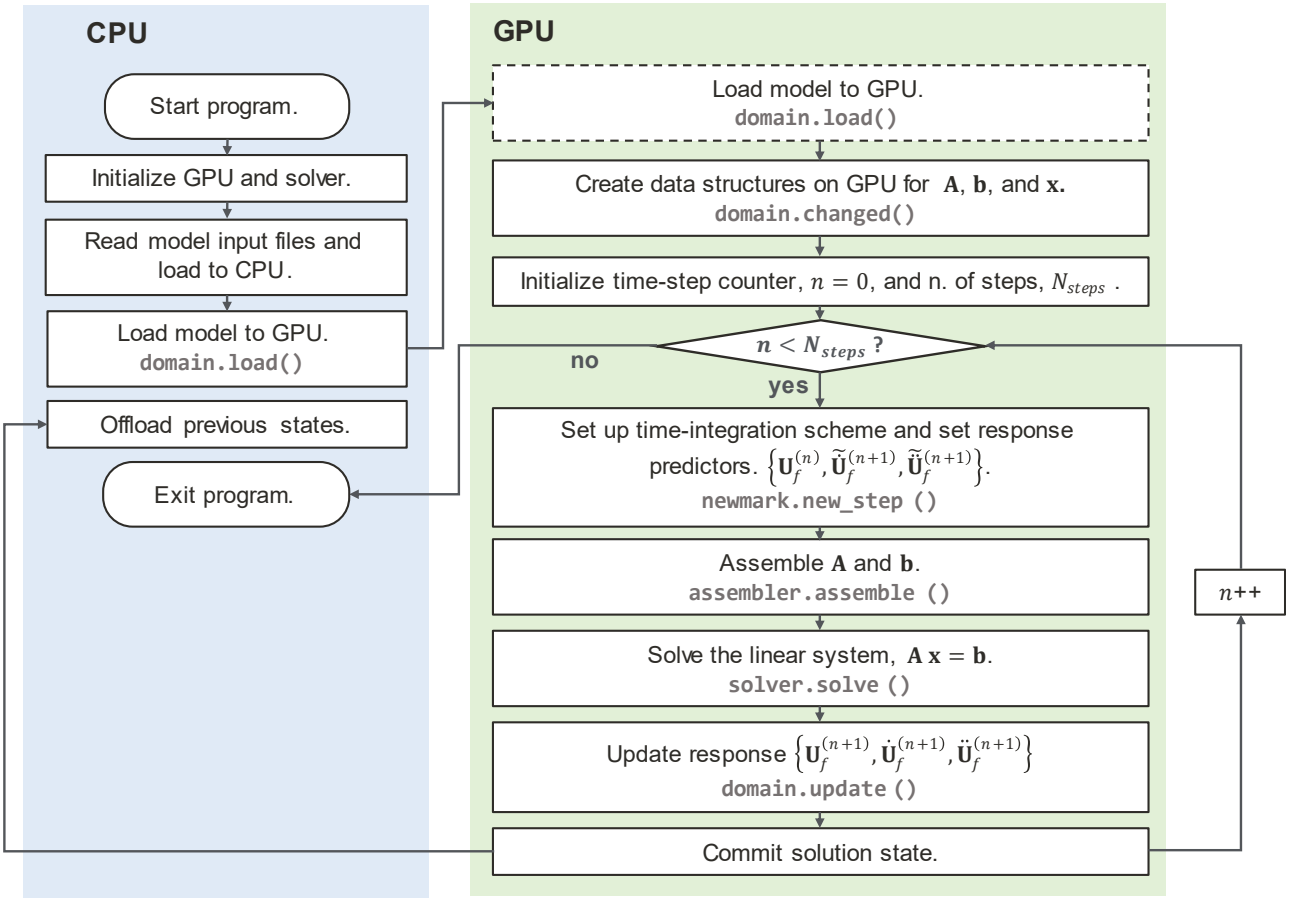

Figure 1. Fully GPU-accelerated elastic FEA.

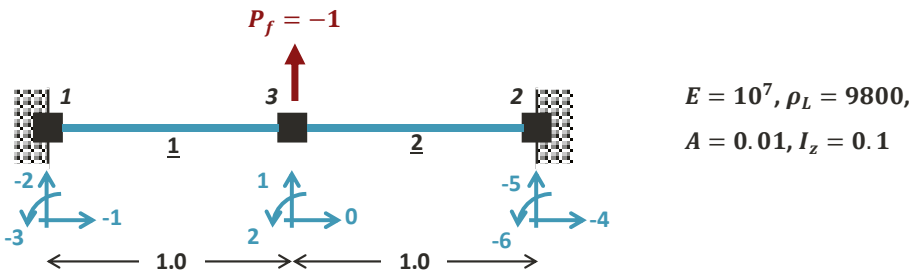

Figure 2. Minimal example to illustrate the GPU-accelerated FEA implementation in CUDA.

2.1 User input

The user defines the instructions on the CPU host on how to port and solve the problem on the GPU device. The program requires the user to define the model in an input text file, set the number of threads to be used during the analysis, and set the solver configuration for the linear system of equations.

Model input file

The user defines the model as a text file using the format in **Figure 3**, where words in **boldface** are mandatory headers and each line after the **nodes** and **elements** header contains the data for an individual node or

element. Note, nodes and elements are assumed to be numbered consecutively starting from 1 in this pilot implementation.

The current implementation of the program estimates a lumped mass matrix derived from the element density. Future work includes adding support for nodal masses.

```
ndm
# NUMBER_OF_DIMENSIONS
2

nodes
# NUMBER_OF_DOFS COORDS FIXITIES LOADS DISPLACEMENTS VELOCITIES ACCELERATIONS
3 0.0 0.0 1 1 1                 # node 1, fixed.
3 0.0 2.0 1 1 1                 # node 2, fixed.
3 0.0 1.0 0 0 0 0.0 -1.0 0.0    # node 3, free and loaded.

elements
# NUMBER_OF_NODES NODE_LIST ELEMENT_TYPE PROPERTIES
2 1 3 1 10000000.0 0.01 0.1 9800.0    # element 1, nodes {1, 3}.
2 3 2 1 10000000.0 0.01 0.1 9800.0    # element 2, nodes {3, 2}.
```

Figure 3. Input file format.

Number of threads

In the CUDA programming model, the programmer defines functions known as kernels that are executed in parallel by a user-specified number of different CUDA threads. These threads are organized into programming hierarchical units called thread blocks and grids. All threads in a block are expected to reside on the same streaming multiprocessor (SM) core. Within NVIDIA GPU architectures, threads in a block are scheduled and executed in groups of 32 threads, known as warps, by the SM.

To optimize efficiency, the user must carefully select the number of threads in a block (Kirk and Hwu, 2016). It is advisable to choose a number of threads per blocks that is a multiple of the warp size, meaning a multiple of 32, and align it with the number of warp schedulers per SM in the GPU device. The number of blocks should also be a multiple of the number of SMs in the device. This minimizes the number of threads that are inactive during the execution of the program.

Solver configuration file

This study uses the AmgX library (Naumov *et al.*, 2015) and its associated C++ application programming interface (API) to iteratively solve the linear system of equations represented by **Eq. (2)** on the GPU. AmgX offers optimized methods for massive parallelism and flexibility to choose between different solvers and preconditioners. Prior to running an analysis, the user specifies the algorithm to be employed, as well as options and parameters to configure that algorithm in a JSON file. For example, the user can specify to solve the problem using the preconditioned conjugate gradient (PCG) method (Barrett *et al.*, 1994; Hestenes and Stiefel, 1952) with 1000 maximum iterations; see **Figure 4**. The PCG method requires a preconditioner (e.g., herein Block Jacobi), which is also defined in the configuration file. More information on how to setup a configuration file is available in the AmgX reference manual (NVIDIA, 2017b).

```
{
    "config_version": 2,
    "determinism_flag": 1,
    "solver": {
        "preconditioner": {
            "error_scaling": 0,
            "print_grid_stats": 1,
            "max_uncolored_percentage": 0.05,
            "algorithm": "AGGREGATION",
            "solver": "BLOCK_JACOBI",
            "smoother": "BLOCK_JACOBI",
            "presweeps": 0,
            "selector": "SIZE_2",
            "coarse_solver": "DENSE_LU_SOLVER",
            "max_iters": 1,
            "postsweeps": 3,
            "min_coarse_rows": 32,
            "relaxation_factor": 0.75,
            "scope": "amg",
            "max_levels": 100,
            "matrix_coloring_scheme": "PARALLEL_GREEDY",
            "cycle": "V",
            "ilu_sparsity_level": 1,
            "coloring_level": 2
        },
        "use_scalar_norm": 1,
        "solver": "PCG",
        "print_solve_stats": 1,
        "obtain_timings": 1,
        "max_iters": 1000,
        "monitor_residual": 1,
        "gmres_n_restart": 30,
        "convergence": "RELATIVE_INI_CORE",
        "scope": "main",
        "tolerance": 1e-3,
        "norm": "L2"
    }
}
```

Figure 4. Example of a solver configuration file using the PCG solver with a Block Jacobi preconditioner.

2.2 Initialize GPU and Solver

The program begins its execution on the CPU after processing the aforementioned input parameters from the user, specifically the model input and solver configuration files. Subsequently, the AmgX library is initialized using the utility API functions detailed in the AmgX Reference Manual (NVIDIA, 2017b), notably *AMGX_initialize()* and *AMGX_initialize_plugins()*. Any errors that may occur during the execution of AmgX are managed using the *AMGX_install_signal_handler()* API function.

Following the initialization of the AmgX library, the program proceeds to read the user-provided configuration file (**Figure 4**). Using this configuration file, it creates an AmgX *Resources* object, which allocates pertinent information about the GPU, and initializes the *Solver*. The final step of the initialization process involves the formation of AmgX objects that represent the matrix $\mathbf{A}$ and vectors $\mathbf{b}$ and $\mathbf{x}$.

2.3 Read and load model in CPU memory

The model input file is processed by the CPU. Two classes, *NodeHost* and *ElementHost*, are defined to collect and organize the information from the input file in the CPU memory. A single instance of *NodeHost* and *ElementHost* is responsible for storing the data representing the entire model. While processing the input, the

host (CPU) assigns numerical labels to the DOFs. Unconstrained DOFs are labelled with positive integers, while constrained-DOF labels are labelled as negative. The numbering process is carried out in the sequence in which nodes are specified in the input file.

2.4 *domain.load()*: Create domain and load model data to the GPU

The model domain is represented with a *Domain* object that contains the spatial representation of the model and the dynamic response at the free DOFs. The domain is initialized, and the data contained in the *NodeHost* and *ElementHost* objects in the CPU memory (i.e., load nodal data, initial nodal response, element connectivity, element properties, etc) is copied to a *Node* and an *Element* object within the *Domain* that store the data in allocated GPU memory.

2.5 *domain.changed()*: Create data structures on GPU

The connectivity and DOF numbering data contained in the *Node* and *Element* objects is used to compute the number of equations or free DOFs, N_f, and the number of non-zero entries in the system stiffness matrix, N_{nz}. The value of N_f defines the size of the allocated memory for the right-hand side vector $\mathbf{b}$ and solution vector $\mathbf{x}$ in **Eq. (2)**. Similarly, N_{nz} defines the amount of memory allocated to store the system matrix $\mathbf{A}$ in compressed sparse row (CSR) format in the GPU memory. The CSR format allows $\mathbf{A}$ to be represented efficiently in memory by storing only the non-zero entries in a linear array (size = N_{nz} values), along with an array containing their corresponding column indices (size = N_{nz} indices), and a separate array of row pointers indicating the start and end indices of each row (size = $N_f + 1$ row pointers).

The computation of N_{nz} is parallelised by having each CUDA thread compute the number of entries in the local stiffness matrix from an element in the model. The partial contribution to N_{nz} of each element is stored in a linear array in the GPU memory and the total sum is executed using reduction and pre-fix sum operations in the CUDA C++ *Thrust* library (NVIDIA, 2018). A similar approach using reduction operations from *Thrust* is used to compute N_f using the number of DOFs associated with each node and its corresponding constraints.

Memory access patterns in the platform are designed such that a thread operates on a single element and adds its contribution to $\mathbf{A}$ and $\mathbf{b}$. Given that some elements will share degrees of freedom, extra memory is also allocated for duplicate element entries in the left-hand side matrix ($\mathbf{A_{dup}}$) and right-hand side vector ($\mathbf{b_{dup}}$) where elements share the same DOF and need to be entered at the same location of the global system matrices. To prepare for filling matrices $\mathbf{A_{dup}}$ and $\mathbf{b_{dup}}$, two arrays of pointers ($\mathbf{A_{map}}$ and $\mathbf{b_{map}}$) map the entries from the local stiffness matrix and load vector of each element to the corresponding memory addresses (represented with the symbol & in C++) in $\mathbf{A_{dup}}$ and $\mathbf{b_{dup}}$; see **Figure 5**.

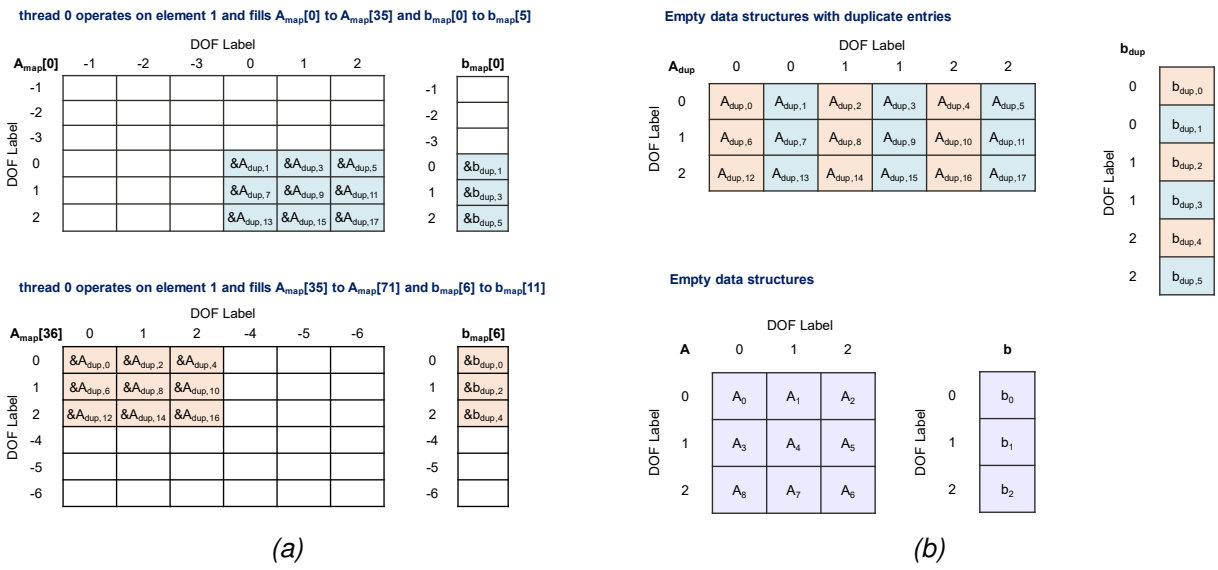

Figure 5. Memory allocation and access patterns for the minimal example: (a) local-to-global matrix and vector maps for each element, and (b) data structures to store left-hand side matrix $\boldsymbol{A}$ and right-hand side vector $\boldsymbol{b}$ with and without duplicate entries.

2.6 *newmark.new_step()*: Setup time-integration scheme

Using the incremental form of Newmark's method, the resulting left-hand side matrix ($\mathbf{A}$) and right-hand side vector ($\mathbf{b}$) in **Eq. (2)** are:

$$\mathbf{A} = \mathbf{K} + \frac{\gamma}{\beta\,\Delta t}\mathbf{C} + \frac{1}{\beta\,\Delta t^2}\mathbf{M}, \tag{3}$$

$$\mathbf{b} = \mathbf{P}_f^{(n+1)} - \mathbf{P}_0^{(n+1)} - \mathbf{K}\,\mathbf{U}_f^{(n+1)} - \mathbf{C}\tilde{\mathbf{U}}_f^{(n+1)} - \mathbf{M}\tilde{\ddot{\mathbf{U}}}_f^{(n+1)}, \tag{4}$$

γ and β = parameters from Newmark's method, and $\tilde{\mathbf{U}}_f^{(n+1)}$ and $\tilde{\ddot{\mathbf{U}}}_f^{(n+1)}$ = velocity and displacement estimates. The solution $\mathbf{x}$ corresponds to the change in displacement response from time step n to time step $n+1$:

$$\mathbf{x} = \Delta\mathbf{U}_f = \mathbf{U}_f^{(n+1)} - \mathbf{U}_f^{(n)}. \tag{5}$$

As part of the setup of the time-integration scheme, the function *newmark.new_step()* preliminary sets the nodal velocities and accelerations in the domain (stored in the *Node* container object in the GPU) to trial quantities $\tilde{\mathbf{U}}_f^{(n+1)}$ and $\tilde{\ddot{\mathbf{U}}}_f^{(n+1)}$, which are response estimates assuming zero increment in displacement:

$$\tilde{\mathbf{U}}_f^{(n+1)} = \left(1 - \frac{\gamma}{\beta}\right)\dot{\mathbf{U}}_f^{(n)} + \Delta t\left(1 - \frac{0.5\gamma}{\beta}\right)\ddot{\mathbf{U}}_f^{(n)}, \tag{6}$$

$$\tilde{\ddot{\mathbf{U}}}_f^{(n+1)} = -\left(\frac{1}{\beta\Delta t}\right)\dot{\mathbf{U}}_f^{(n)} + \left(1 - \frac{0.5}{\beta}\right)\ddot{\mathbf{U}}_f^{(n)}. \tag{7}$$

The *newmark.new_step()* function employs a CUDA kernel to sets the trial values in **Eq. (6)** and **(7)**, with each node being assigned to a separate thread. These trial values are later used by the assembler to compute the right-hand side vector $\mathbf{b}$, as shown in **Eq. (4)**.

2.7 *assembler.assemble()*: Assemble global matrix

The assembly process involves mapping element contributions from local stiffness, damping, and mass matrices to global entries in the system matrix $\mathbf{A}$ and vector $\mathbf{b}$. To ensure parallel execution without race conditions, an intermediate matrix $\mathbf{A_{dup}}$ and a vector $\mathbf{b_{dup}}$, both larger in size than $\mathbf{A}$ and $\mathbf{b}$, respectively, are initially populated with duplicate contributions from elements sharing degrees of freedom, as illustrated in **Figure 5**(a-b). This duplication facilitates the subsequent execution of a CUDA kernel for assembly, with each thread responsible for mapping the contributions from a single element.

Once $\mathbf{A_{dup}}$ and $\mathbf{b_{dup}}$ are assembled, two additional CUDA kernels are invoked to sum and reduce duplicates and fill the corresponding entries in $\mathbf{A}$ and $\mathbf{b}$. During the reduction on $\mathbf{A_{dup}}$ and $\mathbf{b_{dup}}$, each thread is tasked with adding the duplicate entries for a specific non-zero entry in $\mathbf{A}$ and row entry in $\mathbf{b}$.

2.8 *solver.solve()*: Solve system

The system of equations $\mathbf{Ax} = \mathbf{b}$ is solved at each time step using the AmgX *Solver* with the configurations provided by the user; see **Figure 4**. To fully take advantage of the GPU massively parallel resources, this study uses the iterative PCG solver, as it is formulated to require only matrix-vector products that are amenable to the massive parallelism of the GPU. Preconditioning is used to replace the original $\mathbf{Ax} = \mathbf{b}$ with an equivalent set of equations with a better condition number with the aim of less iterations.

2.9 *domain.update()*: Update response and offload to CPU

After solving the linear system of equations, the nodal displacements, velocities, and accelerations are determined based on Newmark constants:

$$\mathbf{U}_f^{(n+1)} = \mathbf{U}_f^{(n)} + \Delta\mathbf{U_f}, \tag{8}$$

$$\dot{\mathbf{U}}_f^{(n+1)} = \tilde{\mathbf{U}}_f^{(n+1)} + \frac{\gamma}{\beta\,\Delta t}\Delta\mathbf{U_f}. \tag{9}$$

$$\ddot{\mathbf{U}}_f^{(n+1)} = \tilde{\ddot{\mathbf{U}}}_f^{(n+1)} + \frac{1}{\beta\,\Delta t^2}\Delta\mathbf{U_f} \tag{10}$$

As with earlier tasks in the program, *domain.update()* is executed in parallel, with each thread handling operations for an individual node. Once the final state is determined, it is committed and transferred back to the CPU. The program then iterates through steps 2.6 to 2.9 for the remaining time steps of the analysis.

3 Summary and conclusions

This study describes a pilot implementation for the linear-dynamic analysis of civil structures in CUDA C++ that leverages GPUs to accelerate computation. The program uses a simple interface where users define the numerical model on the CPU using a model input file, configure the linear-algebra solver using the *AmgX* library, and set the number of CUDA threads. Upon loading the model into CPU memory, the domain is created in GPU memory and the model is copied to the GPU. To estimate the dynamic response, Newmark's method is used to discretize the equations of motion in time and transform them into a linear system of the form $\mathbf{A}\,\mathbf{x} = \mathbf{b}$. To optimize memory use, the CSR format is used to store the non-zero entries in matrix $\mathbf{A}$ and all entries in vector $\mathbf{b}$. The number of non-zero entries and rows, determining the memory allocated for $\mathbf{A}$ and $\mathbf{b}$, is computed in parallel from the contributions of each element later added using sum and reduction operations from the *Thrust* library. Given that some elements will share degrees of freedom, extra memory is also allocated for duplicate element entries ($\mathbf{A_{dup}}$ and $\mathbf{b_{dup}}$). This duplication facilitates the subsequent execution of a CUDA kernel for assembly of $\mathbf{A_{dup}}$ and $\mathbf{b_{dup}}$, with each thread handling the contributions from a single element. Once $\mathbf{A_{dup}}$ and $\mathbf{b_{dup}}$ are assembled, two additional CUDA kernels are invoked to sum and reduce duplicates and fill the corresponding entries in $\mathbf{A}$ and $\mathbf{b}$. Then, the system $\mathbf{A}\,\mathbf{x} = \mathbf{b}$ is solved at every time step using the iterative PCG solver, formulated to require only matrix-vector products that are amenable to the massive parallelism of the GPU. Once a solution has been found, the domain is updated and the dynamic response at the free DOFs is offloaded to the CPU. Notably, the proposed pilot implementation achieves massive parallelism, enabling the GPU code to operate simultaneously on each node or element in the model. This design ensures that computational time for the assembly and update tasks remains nearly independent of the number of DOFs.

Although speedups can be significant for the assembly and update tasks, the program still needs to be refined to maximize speedups on the GPU, as described in Simpson et al. (2023). In particular, several challenges still need to be addressed to capitalize on GPU-driven speedups for nonlinear dynamic analysis, especially the handling of the solver and state determination. Although promising and prevalent in other fields, questions remain on which types of algorithms best translate into efficient GPU acceleration for structural analysis problems (e.g., in terms of the time-stepping integration scheme and solver).

Ongoing work by the authors is currently underway to extend this GPU acceleration to include nonlinear response with heterogeneous elements and inelastic material formulations. It is expected that such work may deviate substantially from the traditionally formulated approaches described herein for this pilot implementation. Future work will compare other solvers and time integration schemes, including explicit or semi-implicit methods. A baseline for comparison of multicore CPU to GPU computing would give an effective comparison of speedups; i.e., speedups against single-core CPUs may look more impressive than multicore CPUs.

4 Acknowledgements

This research was supported by National Science Foundation (NSF) under grant number CMMI-2145665, titled CAREER: Accelerating Real-time Hybrid Physical-Numerical Simulations in Natural Hazards Engineering with a Graphics Processing Unit (GPU)-driven Paradigm. The findings, opinions, recommendations, and conclusions in this paper are those of the authors alone and do not necessarily reflect the views of others, including the sponsors.

5 References

Amdahl, G.M. (1967), "Validity of the single processor approach to achieving large scale computing capabilities", *AFIPS '67 (Spring): Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, Association for Computing Machinery, Atlantic City, New Jersey, pp. 483–485, doi: https://doi.org/10.1145/1465482.1465560.

Barrett, R., Berry, M., Chan, T.F., Demmel, J., Donato, J.M., Dongarra, J., Eijkhout, V., *et al.* (1994), *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, 2nd ed., SIAM, Philadelphia, PA.

Bartezzaghi, A., Cremonesi, M., Parolini, N. and Perego, U. (2015), "An explicit dynamics GPU structural solver for thin shell finite elements", *Computers and Structures*, Elsevier Ltd, Vol. 154, pp. 29–40, doi: 10.1016/j.compstruc.2015.03.005.

Chiang, K.N. and Fulton, R.E. (1990), "Concepts and Implementation of Parallel Finite Element Analysis", *Computers & Structures,* Vol. 36 No. 6, pp. 1039–1046, doi: https://doi.org/10.1016/0045-7949(90)90211-J.

Elgamal, A., Yan, L., Yang, Z. and Conte, J.P. (2008), "Three-Dimensional Seismic Response of Humboldt Bay Bridge-Foundation-Ground System", *Journal of Structural Engineering*, Vol. 134 No. 7, pp. 1165–1176, doi: 10.1061/ASCE0733-94452008134:71165.

Georgescu, S., Chow, P. and Okuda, H. (2013), "GPU Acceleration for FEM-Based Structural Analysis", *Archives of Computational Methods in Engineering*, June, doi: 10.1007/s11831-013-9082-8.

Gimenez, J.M., Ramajo, D.E., Márquez Damián, S., Nigro, N.M. and Idelsohn, S.R. (2017), "An assessment of the potential of PFEM-2 for solving long real-time industrial applications", *Computational Particle Mechanics*, Springer International Publishing, Vol. 4 No. 3, pp. 251–267, doi: 10.1007/s40571-016-0135-2.

Hestenes, M.R. and Stiefel, E. (1952), "Methods of Conjugate Gradients for Solving Linear Systems", *Journal of Research of the National Bureau of Standards*, Vol. 49 No. 6, pp. 409–436, doi: 10.6028/jres.049.044.

Jeremić, B. and Jie, G. (2008), "Parallel Soil–Foundation–Structure Interaction Computations", in Papadrakakis, M., Charmpis, D.C., Tsompanakis, Y. and Lagaros, N.D. (Eds.), *Computational Structural Dynamics and Earthquake Engineering: Structures and Infrastructures Book Series*, 1st ed., Vol. 2, CRC Press, pp. 427–446, doi: 10.1201/9780203881637-38.

Johnsen, S.F., Taylor, Z.A., Clarkson, M.J., Hipwell, J., Modat, M., Eiben, B., Han, L., *et al.* (2015), "NiftySim: A GPU-based nonlinear finite element package for simulation of soft tissue biomechanics", *International Journal of Computer Assisted Radiology and Surgery*, Springer Verlag, Vol. 10 No. 7, pp. 1077–1095, doi: 10.1007/s11548-014-1118-5.

Joldes, G.R., Wittek, A. and Miller, K. (2010), "Real-time nonlinear finite element computations on GPU - Application to neurosurgical simulation", *Computer Methods in Applied Mechanics and Engineering*, Vol. 199 No. 49–52, pp. 3305–3314, doi: 10.1016/j.cma.2010.06.037.

Kang, D.K., Kim, C.W. and Yang, H.I. (2014), "GPU-based parallel computation for structural dynamic response analysis with CUDA", *Journal of Mechanical Science and Technology*, Korean Society of Mechanical Engineers, Vol. 28 No. 10, pp. 4155–4162, doi: 10.1007/s12206-014-0928-2.

Kirk, D.B. and Hwu, W.W. (2016), *Programming Massively Parallel Processors: A Hands-on Approach*, 3rd ed., Morgan Kaufmann, Waltham, MA.

Komatitsch, D., Michéa, D. and Erlebacher, G. (2009), "Porting a high-order finite-element earthquake modeling application to NVIDIA graphics cards using CUDA", *Journal of Parallel and Distributed Computing*, Vol. 69 No. 5, pp. 451–460, doi: 10.1016/j.jpdc.2009.01.006.

Lu, X. and Guan, H. (2017), *Earthquake Disaster Simulation of Civil Infrastructures: From Tall Buildings to Urban Areas*, 1st ed., Springer Singapore, doi: https://doi.org/10.1007/978-981-10-3087-1.

Mackerle, J. (1996), "Implementing finite element methods on supercomputers, workstations and PCs: a bibliography (1985-1995)", *Engineering Computations*, Vol. 13 No. 1, pp. 33–85, doi: https://doi.org/10.1108/02644409610110985.

Mackerle, J. (2003), "FEM and BEM parallel processing: Theory and applications - A bibliography (1996-2002)", *Engineering Computations*, Emerald Group Publishing Ltd., Vol. 20 No. 3–4, pp. 436–484, doi: 10.1108/02644400310476333.

Mackerle, J. (2004), "Object-oriented programming in FEM and BEM: A bibliography (1990-2003)", *Advances in Engineering Software*, Elsevier Ltd, Vol. 35 No. 6, pp. 325–336, doi: 10.1016/j.advengsoft.2004.04.006.

Mafi, R. and Sirouspour, S. (2014), "GPU-based acceleration of computations in nonlinear finite element deformation analysis", *International Journal for Numerical Methods in Biomedical Engineering*, Wiley-Blackwell, Vol. 30 No. 3, pp. 365–381, doi: 10.1002/cnm.2607.

McCallen, D., Tang, H., Wu, S., Eckert, E., Huang, J. and Petersson, N.A. (2022), "Coupling of regional geophysics and local soil-structure models in the EQSIM fault-to-structure earthquake simulation framework", *The International Journal of High Performance Computing Applications*, Vol. 36 No. 1, pp. 78–92, doi: 10.1177/10943420211019118.

McKenna, F. and Fenves, G.L. (2000), "An Object-Oriented Software Design for Parallel Structural Analysis", *Advanced Technology in Structural Engineering*, doi: 10.1061/40492(2000)30.

McKenna, F.T. (1997), *Object-Oriented Finite Element Programming: Frameworks for Analysis, Algorithms and Parallel Computing*, Dissertation, University of California, Berkeley, Berkeley.

Motley, M.R., Wong, H.K., Qin, X., Winter, A.O. and Eberhard, M.O. (2016), "Tsunami-Induced Forces on Skewed Bridges", *Journal of Waterway, Port, Coastal, and Ocean Engineering*, American Society of Civil Engineers (ASCE), Vol. 142 No. 3, doi: 10.1061/(asce)ww.1943-5460.0000328.

Naumov, M., Arsaev, M., Castonguay, P., Cohen, J., Demouth, J., Eaton, J., Layton, S., *et al.* (2015), "AMGX: A library for GPU accelerated algebraic multigrid and preconditioned iterative methods", *SIAM Journal on Scientific Computing*, Society for Industrial and Applied Mathematics Publications, Vol. 37 No. 5, pp. S602–S626, doi: 10.1137/140980260.

NVIDIA. (2017a), "CUDA Toolkit 9.0", NVIDIA, September.

NVIDIA. (2017b), "AMGX Reference Manual", NVIDIA, October.

NVIDIA. (2018), *Thrust Quick Start Guide*.

O'Reilly, O., Yeh, T.Y., Olsen, K.B., Hu, Z., Breuer, A., Roten, D. and Goulet, C.A. (2022), "A High-Order Finite-Difference Method on Staggered Curvilinear Grids for Seismic Wave Propagation Applications with Topography", *Bulletin of the Seismological Society of America*, Seismological Society of America, Vol. 112 No. 1, pp. 3–22, doi: 10.1785/0120210096.

Posey, S. and Wang, P. (2012), "GPU progress in sparse matrix solvers for applications in computational mechanics", *50th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition*, doi: 10.2514/6.2012-562.

Roten, D., Cui, Y., Olsen, K.B., Day, S.M., Withers, K., Savran, W.H., Wang, P., *et al.* (2016), "High-Frequency Nonlinear Earthquake Simulations on Petascale Heterogeneous Supercomputers", *SC16, Supercomputing Conference*, IEEE Press, Salt Lake City, UT, USA, pp. 957–968, doi: 10.1109/SC.2016.81.

Simpson, B.G., Zhu, M., Seki, A. and Scott, M.H. (2023), "Challenges in GPU-Accelerated Nonlinear Dynamic Analysis for Structural Systems", *Journal of Structural Engineering*, American Society of Civil Engineers, Vol. 149 No. 3, p. 04022253, doi: 10.1061/JSENDH.STENG-11311.

Taylor, Z.A., Cheng, M. and Ourselin, S. (2008), "High-speed nonlinear finite element analysis for surgical simulation using graphics processing units", *IEEE Transactions on Medical Imaging*, Vol. 27 No. 5, pp. 650–663, doi: 10.1109/TMI.2007.913112.

TBI. (2017), "Guidelines for Performance-Based Seismic Design of Tall Buildings", Pacific Earthquake Engineering Center, Berkeley, CA, United States.

Topping, B.H. V. and Khan, A.I. (1996), *Parallel Finite Element Computations*, Saxe-Coburg Publications, Edinburgh.

Wilson, E.L. (2010), *Three Dimensional Static and Dynamic Analysis of Structures: A Physical Approach With Emphasis on Earthquake Engineering*, 4th ed., Computers & Structures Inc., Berkeley, CA.

Zhu, M. and Scott, M.H. (2014), "Modeling fluid-structure interaction by the particle finite element method in OpenSees", *Computers and Structures*, Vol. 132, pp. 12–21, doi: 10.1016/j.compstruc.2013.11.002.