Challenges in GPU-Accelerated Nonlinear Dynamic Analysis for Structural Systems

Barbara G. Simpson, Ph.D., M.ASCE[1]; Minjie Zhu, Ph.D.[2]; Akiri Seki[3]; and Michael Scott, Ph.D.[4]

Abstract: Many numerical simulation methods, such as finite-element analysis, were originally formulated to run serially or in parallel on central processing units (CPUs). However, computer engineering has seen a paradigm shift toward massive parallelism using graphics processing units (GPUs), which have become the default accelerators in many data-driven scientific disciplines outside of civil engineering. This state-of-the-art review highlights the challenges and practicalities of GPU-accelerating nonlinear dynamic analyses for civil structural problems. To demonstrate the feasibility of a fully GPU-accelerated finite-element analysis, a GPU-based program for linear-elastic dynamic analysis was implemented, where all stages of the analysis were ported to the GPU. Observed speedups were 115 times that of an equivalent CPU-driven analysis for 10^6 model degrees of freedom (dof). Importantly, the computational time for the assembly and update levels of the analyses were nearly independent of the number of dof. High-resolution simulations of complex structures can be computationally expensive, but these results and advances in other fields suggest that some levels of the finite-element analysis of civil structures could be accelerated using GPUs at increased model resolution with little increase in computational cost, demonstrating the potential for GPU-accelerated computing. However, compared to other GPU-accelerated finite-element analysis applications, the dynamic analysis of civil structures is subject to unique challenges that need to be addressed before GPU acceleration can be fully realized. Aspects of simulating the response of civil structures considering nonlinear response under extreme loading may not be immediately amenable to GPU acceleration; e.g., the use of many differing element formulations within a model, potential for inelastic response and varying degrees of nonlinearity across elements, and traditional reliance on implicit integration schemes with direct solvers. The shift to GPUs is part of a larger movement toward specialized hardware using fine-grained parallelism, and structural engineers need to address these challenges as these emerging technologies become more prevalent. **DOI: 10.1061/JSENDH.STENG-11311.**

Introduction

Computational bottlenecks must be overcome to realize higher-fidelity models of civil structures subjected to natural hazards. Finite-element-based structural analysis is a common tool used by a wide range of engineering disciplines (Bathe 1996; Hughes 1987; Zienkiewicz and Taylor 1989). However, the resolution of structural models, in terms of both model size and mesh refinement, can be limited by computational cost, resulting in an ever-increasing need for acceleration. For example, realistic three-dimensional models can take hours or days to run one analysis, e.g., as observed for tsunami loading on bridges (Motley et al. 2016), bridge-foundation-ground interaction (Elgamal et al. 2008), tall building response (Lu and Guan 2017), among others. Regional-scale modeling, uncertainty propagation, optimization, and many "interaction"-type problems, e.g., soil-structure interaction (McCallen et al. 2022) or fluid-structure interaction (Gimenez et al. 2017; Zhu and Scott 2014), often suffer from long run times and require large-scale computing resources.

Parallel processing on central processing units (CPUs) has long been used to accelerate finite-element analyses (FEA). Multicore parallel computing on CPUs partitions the spatial domain to run on multiple processors (Mackerle 1996, 2003, 2004; McKenna 1997; Topping and Khan 1996). The parallelized equations are then solved using a special parallel equation solver. However, when the domain is run in parallel this way, the subdomains split across the cores are still coupled, necessitating intracore communication between parts of the analysis running on different cores. This communication eventually slows down the analysis as more and more cores beyond the optimum number of cores are utilized, as observed in Jeremic and Jie (2008). By Amdahl's law (Amdahl 1967), using more processors is limited by the portions of the problem that cannot be parallelized, e.g., intracore communication or data management housekeeping. This communication causes the total performance of parallel CPU-based analyses to inevitably plateau; i.e., increasing the number of CPU cores does not always result in faster simulations.

Driven by demands for ever more realistic graphics rendering in the 1980s, the performance of modern GPUs has surpassed that of CPUs (Borkar et al. 2005; Sutter 2005) in efforts to display thousands to millions of pixels simultaneously. Originally developed as specialized hardware for scene rendering, GPUs have since become the default accelerators for applications in deep learning, robotics, self-driving vehicles, virtual or augmented reality, and, now, supercomputing (Owens et al. 2005). High-performance computing (HPC) applications in quantum chemistry, molecular dynamics (Anderson et al. 2008; Chen et al. 2016; Sunarso et al. 2010), climate modeling (Nyland et al. 2007), computational fluid dynamics

[1]Assistant Professor, School of Civil and Environmental Engineering, Stanford Univ., Stanford, CA 94305 (corresponding author). ORCID: https://orcid.org/0000-0002-3661-9548. Email: bsimpson@stanford.edu

[2]Research Associate, School of Civil and Construction Engineering, Oregon State Univ., Corvallis, OR 97331. Email: zhum@oregonstate.edu

[3]Graduate Student Researcher, School of Civil and Environmental Engineering, Stanford Univ., Stanford, CA 94305. ORCID: https://orcid.org/0000-0002-7736-7375. Email: sekia@oregonstate.edu

[4]Professor, School of Civil and Construction Engineering, Oregon State Univ., Corvallis, OR 97331. ORCID: https://orcid.org/0000-0001-5898-5090. Email: Michael.Scott@oregonstate.edu

Note. This manuscript was submitted on January 21, 2022; approved on August 24, 2022; published online on December 22, 2022. Discussion period open until May 22, 2023; separate discussions must be submitted for individual papers. This paper is part of the ***Journal of Structural Engineering***, © ASCE, ISSN 0733-9445.

(CFD) (Appleyard and Drikakis 2011; Brandvik and Pullan 2007; Corrigan et al. 2011; Kampolis et al. 2010), and healthcare imaging (Sylwestrzak et al. 2017) have since been GPU-accelerated (NVIDIA 2015; Snell and Segervall 2017).

Even though seismic wave propagation (Komatitsch et al. 2009; O'Reilly et al. 2022; Roten et al. 2016; Zhou et al. 2013) and biomechanics (Johnsen et al. 2015; Joldes et al. 2010; Mafi 2013; Taylor et al. 2008) have a long history of massively parallel computing on GPUs, the nonlinear dynamic analysis of civil structures is unique due to varied element formulations and potential for inelastic response. While progress has been made to GPU-accelerate simulations of earthquake fault rupture (Komatitsch et al. 2009; Kusakabe et al. 2021; Yamaguchi et al. 2020; Zhou et al. 2013) and ocean fluid dynamics (Tavakkol and Lynett 2017), there has been slower progress in GPU-accelerating analyses for civil structures. This lack of progress can be attributed to the prevalence of different element formulations and nonlinearities, coupled equations of motion, implicit integration schemes, and reliance on direct solvers, which are not immediately amenable to fully parallel acceleration on GPUs.

Although the theory for FEA is mature, analyzing civil structures with more modern computer architectures, e.g., like GPUs, has not yet materialized. Scientific computing using GPUs is only part of a larger movement toward specialized hardware offering parallel processing optimized for specific tasks, and GPU-like types of computing will become increasingly prevalent in any field using numerical simulations, including the structural engineering discipline. Foreseeing the role that GPUs will play in the future, national and international universities and laboratories have sought to enable GPU-driven exascale deployment (Alexander et al. 2020; Kothe et al. 2019; Siegel et al. 2020); e.g., fault-to-structure regional simulations (McCallen et al. 2021a, b; Roten et al. 2016). New technologies in HPC—along with an improved understanding of earthquake hazards, risk, and building performance—will only continue to emerge.

Yet, there are barriers to the widespread development of modern FEA applications that utilize GPUs and other acceleration hardware. Although structural engineers have long had access to extreme-scale HPC resources for nonlinear dynamic analysis [FEMA 283 (FEMA 1996)], there is still little understanding of the types of algorithms that translate into efficient GPU acceleration for differing applications (Leung et al. 2010; Peterson et al. 2018). Moreover, a learning curve must be overcome to fully leverage GPUs. As new technologies become available, resources need to be developed so that high-fidelity computation becomes more accessible at reduced computational cost in natural hazards engineering and research, education, and practice. The intent of this review is to provide guidance for the future acceleration of such applications using the GPU acceleration of civil structure problems as an example.

This paper presents a high-level picture of prior work on accelerating FEA for structural and solid mechanics problems via GPUs. Emphasis is placed on structural analysis problems in earthquake engineering, but the methods can be more broadly applied to natural hazards engineering. The architecture and unique programming model of the GPU are described. Existing work on GPU acceleration of FEA has focused on the acceleration of the FEA solver, but progress has been made on accelerating all stages of FEA, including state determination, assembly, and the time-integration scheme. Challenges are highlighted in porting the dynamic analyses of civil structures to the massively parallel architecture of GPUs, including coupled equations of motion, heterogeneous element formulations, and inelastic constitutive laws. To show the potential of GPU acceleration, a demonstration study illustrates that potential speedups for a linear-elastic analysis depend on the number of degrees of freedom, as observed by other disciplines. The aforementioned challenges still need to be addressed to assess the performance of GPU-accelerated nonlinear dynamic analyses incorporating nonlinear geometric and material effects along with varied element formulations.

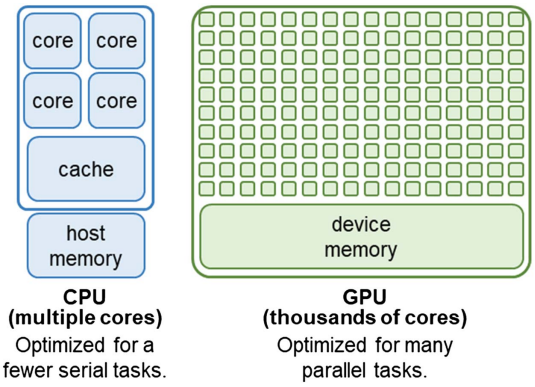

Fig. 1. CPU versus GPU.

Scientific Computing and Advances in Graphics Processing Units

Due to power and heat dissipation constraints, computer architectures have moved from increasing the performance of individual cores to increasing the number of cores on a chip; see Fig. 1. In theory, this evolution has led to CPUs that divide work in time (by completing serial tasks as fast as possible, low *latency*) and GPUs that divide work in space (by completing as many tasks as possible, high *throughput*). This massive parallelism allows GPUs to execute many tasks (thousands) simultaneously. While CPUs aim for speed, i.e., completing a task as fast as possible, GPUs complete as many tasks as possible by dividing the work among a large number of *threads*. Each thread may not be as fast as a single CPU core, but the communication between GPU threads is lightweight and allows thousands of threads to execute instructions at the same time, resulting in more floating-point operations per second (FLOPs) (Owens et al. 2005). The GPU itself can range from relatively cheap, single-precision (SP) GPUs (often used in machine learning) with higher memory bandwidth to high-performance double-precision GPUs (promoted for scientific computing).

GPUs exploit fine-grain single-instruction-multiple-thread (SIMT) parallelism, where multiple threads are processed by a single instruction concurrently. Thus, GPUs perform best for problems divisible into a large number of small tasks that require little-to-no communication (i.e., *embarrassingly parallel*). For example, independent operations, like matrix multiply, can be easily made parallel and are extremely fast on GPUs.

Although this paper focuses on GPUs, similar flavors of single-instruction-multiple-data (SIMD) parallelism also exist, which applies the same instructions to multiple data: Cray (Carey et al. 1988; Ting et al. 2004), CPU vector processing units (e.g., Intel MMX/SSEn/AVX), PowerPC AltiVec, ARM Advanced SIMD, Intel's Xeon Phi (discontinued), and Tensor Processing Units (TPUs). In many ways, the modern GPU can be viewed as a successor of the classic Cray supercomputers, and methods of vectorizing FEA can be gained from the Cray architectures (Carey et al. 1988; Ting et al. 2004). To maximize performance, the most recent AMD accelerated processing units (APUs) integrate CPU-GPU combinations on a single chip. Intel has also proposed XPUs with a

unified programming model for any hardware architecture [CPU, GPU, field programmable gate arrays (FPGA)] that could streamline the next generation of specialized heterogeneous hardware.

General-Purpose GPU Programming

General-purpose GPUs (GPGPUs) (Owens et al. 2005; Rumpf and Strzodka 2005) are GPUs programmed for purposes other than their original application for graphics processing. Nevertheless, there are significant differences between CPU and GPU hardware architectures, resulting in fundamentally different data structures and processes at the programming level.

For early GPUs, operations were programmed manually by mapping scientific calculations to graphics manipulations. After 2006, Nvidia's Compute Unified Device Architecture (CUDA) (NVIDIA 2019) programming language allowed NVIDIA GPUs to be more easily programmed for purposes other than graphics rendering. The Open Computing Language (OpenCL) is an alternative vendor-independent cross-platform language for parallel programming on a diverse set of accelerators, e.g., NVIDIA architectures, AMD and Intel GPUs, and FPGAs, among others. Both CUDA and OpenCL enable a C-like programming experience to run code in parallel on GPUs. Of the two, CUDA remains the most well-supported, providing a development environment equipped with numerous libraries.

Library packages have also been developed to accelerate existing code to run on GPUs at the cost of degraded performance. In OpenACC, directives are added to CPU code to instruct portions of the code to run in parallel on the GPU. Some have found that speedups using OpenACC have been comparable to CUDA with lower development and higher maintainability costs (Kusakabe et al. 2021; Yamaguchi et al. 2020). Other abstractions of low-level HPC access have also been proposed for improved portability of existing codes across platforms, e.g., Intel's OneAPI and RAJA C++ libraries (Beckingsale et al. 2019).

Supercomputing

Although affordable GPUs are widely available on local desktop or laptop machines, supercomputing is needed to contend with large or high-resolution models, e.g., for regional urban environments (O'Reilly et al. 2022) or scale-resolving CFD (Gorobets and Bakhvalov 2022). For example, modeling end-to-end fault-to-structure simulations, including fault rupture, seismic wave propagation, and structural response, requires frequencies to be resolved at 1–2 Hz for ground motion simulations to 5–10 Hz for modeling stiff infrastructure (Cui et al. 2013). However, the needed computation for end-to-end regional simulations is beyond current HPC capabilities, as billions of computations are needed to resolve frequencies relevant to both the ground motion and structural response (McCallen et al. 2021a, b), requiring exascale computing.

To achieve greater performance and energy efficiency, the current generation of petascale machines combine many-core processors, e.g., CPUs, with massively parallel accelerators, e.g., GPUs, to perform quadrillions of calculations each second. The potential for GPU-based exascale supercomputers to run on billions of cores (Alexander et al. 2020) could resolve higher frequencies faster, rendering high-fidelity simulations in civil engineering tractable. As zettascale computing is also on the horizon after exascale computing, hardware relying on heterogeneous hardware will only continue to evolve, and structural engineers must be able to adapt numerical methods and parallel algorithms to benefit from the next generation of supercomputers.

Performance on modern supercomputers is driven by scalability and parallelism of the analysis, along with optimized memory access, reduced memory consumption, efficient scheduling, load balancing (McCallen et al. 2021a), hidden latencies (Roten et al. 2016), and portability to allow for a broad range of architectures (such as many-core CPUs and GPUs from various vendors) (Gorobets and Bakhvalov 2022). The lower-level building blocks of FEA (e.g., data structures, solvers, algorithms) affect accuracy, scalability, and parallelism (Brown et al. 2022). Explicit integration schemes with iterative solvers tend to have better scalability compared to implicit integration schemes with direct solvers depending on the model size and extent of nonlinear response. Similarly, structured mesh using stencils can more easily use GPU resources (described subsequently) (Cui et al. 2013). Exascale deployment also depends on large degrees of massive parallelism, requiring efficient workflows (Cui et al. 2013; McCallen et al. 2021a; Roten et al. 2016) with high-performance I/O operations (Byna et al. 2020) and data management and compression for huge datasets (hundreds of TBs) (Lindstrom 2014).

To date, exascale efforts in end-to-end regional simulations have focused on partitioning the geophysical domain, rather than the structural domain, e.g., using pencil-shaped subdomains of the ground with each structural model assigned to a single core (McCallen et al. 2021a).

GPU Execution Model

As GPUs have an unusual programming model, knowledge of the GPU execution model is often needed to accelerate FEA computations; see Table 1. Herein, the CUDA terminology will be used to describe the GPU architecture, but the concepts are widely applicable to other SIMD processing. CUDA uses abstractions, in terms of a hierarchy of thread groups, shared memory, and barrier synchronization, to partition the problem into coarse subproblems solved independently and fine subproblems solved cooperatively in parallel.

The GPU architecture implies a *for* loop where each iteration of the loop runs simultaneously via the threads. The CUDA code contains a *host* code running on the CPU and a *device* code running on the GPU. The CPU is needed to initialize, organize memory, and interact with the data on the GPU. The GPU executes a basic parallel function, the *kernel*, callable by the CPU, that defines the parallel computations of the threads (the kernel is executed N times in parallel by N threads).

The threads in a kernel are organized by a *grid* of thread *blocks* used to define the location of the thread inside the memory hierarchy of the GPU; see Figs. 2 and 3 for widely available schematics of the memory hierarchy. All threads within a thread block can communicate and synchronize with each other. The thread blocks are subdivided into groups of threads called *warps* typically composed of 32 threads.

Table 1. Useful definitions

Term	Description
Host	The CPU.
Device	The GPU.
Kernel	Function executed in parallel by an array of threads on the device.
Grid	Set of thread blocks that can be executed independently.
Thread Block	Set of threads with common access to shared memory whose execution can be synchronized.
Warp	Group of 16 or 32 threads executed concurrently.

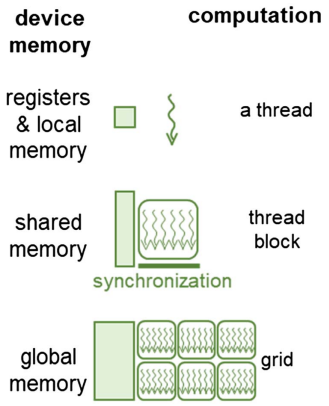

Fig. 2. GPU thread organization.

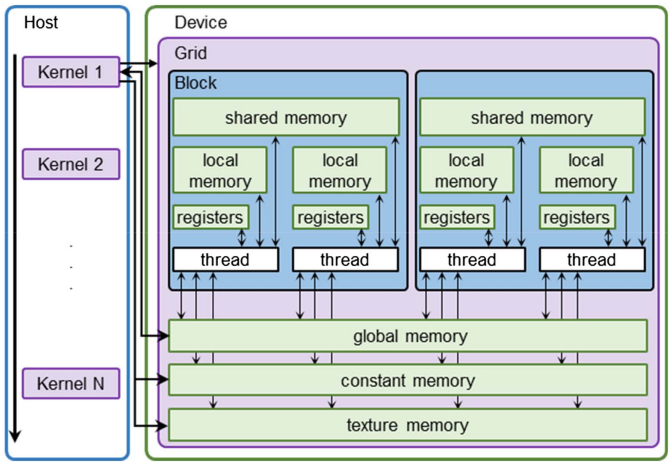

Fig. 3. CUDA memory model.

All threads in a warp must execute the same instructions in lockstep at the same time (in parallel). Thus, thread divergence can occur when the threads within a warp follow different instructions, causing the different paths to be executed serially. For example, if/else conditional statements branch to have different instructions; e.g., when handling different elements or varying degrees of inelastic status (Yang et al. 2014). Since the "if" and "else" branches handle different instructions, the threads assigned to a warp handle each branch serially, resulting in loss of performance.

GPU Memory

CUDA allows developers to extend the C language to identify each thread, its blocks, and organization and transfer of data across the GPU memory hierarchy. The CPU (the host) and GPU (the device) maintain their own memory. Information between thread blocks and data exchange between the CPU and GPU is synchronized through *global memory*, which is off-chip and slow; access to global memory on the GPU is not cached and has large latency. In addition, threads have private *local memory*, which is also off-chip and slow.

In contrast, threads within a thread block can efficiently share and synchronize data through *shared memory*, which is on-chip and whose resources are shared across the warps assigned to the block and is relatively fast and of high bandwidth, if limited in size. As access to shared memory is defined by banks, bank conflicts can occur if two threads executing the same instruction attempt to access the same bank, resulting in serial execution. Shared memory is beneficial for applications where neighboring threads share common data, such as algorithms executed over regular grids, but is unlikely to be beneficial for an unstructured mesh where neighboring threads do not generally share the same data.

Each thread also has a *register*, which can be used to store frequently accessed variables quickly and is limited in size. The GPU also has access to read-only *constant memory* and *texture memory*. The types of available memory, including its optimization, are described extensively in NVIDIA (2013).

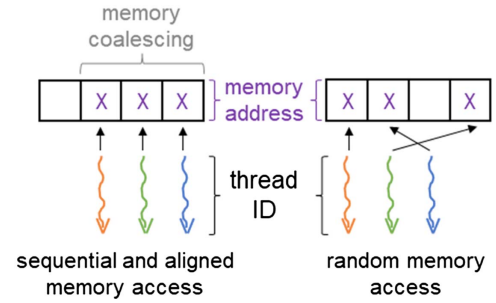

Fig. 4. Memory coalescence.

Performance Considerations

Precision (single 32-bit versus double 64-bit precision), data size, and memory access can all influence GPU performance, which can vary depending on the age/type of the GPU hardware. For example, double precision requires twice the memory as single precision with possible decreases in performance. Yamaguchi et al. (2020) proposed custom low-precision arithmetic for GPUs; e.g., using 21-bit data to provide the accuracy needed for scientific calculations with less memory. Although (Georgescu et al. 2013) compared GPU-based implementations of structural mechanics, most cited literature used single-precision GPUs (Joldes et al. 2010; Komatitsch et al. 2009; Mafi 2013; Taylor et al. 2008) when improved, double-precision GPUs with more memory are now available, as used by Bartezzaghi et al. (2015); all implementations prior to CUDA 1.3 only supported single-precision GPUs.

Generally, available memory on the GPU can be a limiting factor in acceleration (Bartezzaghi et al. 2015; Taylor et al. 2008). Delays also occur every time the GPU exchanges information with global memory, which is large but off-chip and low bandwidth, which can be particularly challenging for heterogeneous CPU-GPU computing environments. Efficient memory access also needs to be sequential and aligned [i.e., *coalesced* (Inoue 2015; Kirk and Hwu 2013)]; see Fig. 4. Coalescence refers to when the threads in a warp are organized consecutively with the corresponding address in memory, allowing for the threads in that warp to access memory simultaneously. Memory access will be serial if the access pattern is not sequential, is sparse, or is misaligned.

"Zero padding" has been used to obtain memory alignment in multiples of 16 (a half-warp), which is optimal in CUDA (Komatitsch et al. 2009). Unchanging parameters can also be hard-coded in constant or texture memory on the GPU, which are read-only but also limited in size (Bartezzaghi et al. 2015; Zhou et al. 2013). Register spilling, when the number of variables that need to be stored is more than the available registers, can also result in performance drops, as the GPU is forced to use local

memory, which has latencies similar to global memory (Bartezzaghi et al. 2015; Zhou et al. 2013).

There are many other performance considerations for GPUs; an in-depth discussion is presented in Kirk and Hwu (2013). For efficient programming on the GPU, any analysis needs to: (1) determine a data flow to minimize local/global memory access and data transfers between the host and device, (2) tune execution to maximize performance (e.g., in terms of the number of threads per block, blocks per grid, and overlapping latencies of memory transfers with useful computation), (3) avoid parallel limitations, like deadlock, race conditions, and load imbalance, (4) maintain coalesced access to global memory, (5) balance the amount of computation with enhanced speed of simultaneous redundant calculations, (6) avoid branches and thread execution divergences, and (7) consider shared memory access patterns (to avoid bank conflicts).

Accelerating Finite-Element Analysis

Advances in GPU hardware and software (i.e., the CUDA programming language has become very robust) provide new HPC opportunities for civil engineering applications. On CPUs, many algorithms for single-core computing need little modification to run on multicore CPUs (Baugh and Sharma 1994; Hajjar and Abel 1988; Kumar and Adeli 1995; Mackerle 2003; Santiago and Law 1996) because of the relatively quick data exchanges between CPU cores with dedicated transistors supporting branch prediction and caching. In contrast, the massive parallelism of GPUs makes directly porting existing algorithms difficult, inefficient, or impossible.

Despite several decades of work in parallel finite-element algorithms for HPC (Topping and Khan 1996), modern FEA software (which was written to run on CPUs) must conform to massive parallelism to fully leverage GPUs. The typical levels of FEA in structural dynamics are shown in Fig. 5(a).

1. The response in terms of displacements $\boldsymbol{U}_k$, velocities $\dot{\boldsymbol{U}}_k$, and accelerations $\ddot{\boldsymbol{U}}_k$ at the beginning of the time step k is known.
2. The state of the elements due to this motion is calculated, as defined by the constitutive law.
3. Based on the element states, the coefficient matrix $\boldsymbol{A}$, and right-hand side vector $\boldsymbol{b}$ are assembled.
4. The linear system of equations, $\boldsymbol{Ax} = \boldsymbol{b}$, is solved to obtain the updated motion response. Depending on the integration scheme, e.g., implicit or explicit, iterations defined by a root-finding algorithm, e.g., Newton–Raphson, can be implemented to satisfy the equations of motion to within a tolerance.
5. Upon convergence, the responses $\boldsymbol{U}_{k+1}$, $\dot{\boldsymbol{U}}_{k+1}$, and $\ddot{\boldsymbol{U}}_{k+1}$ for time step $k+1$ are committed before moving on to the following time step.

The most expensive computations occur during the state determination (step 2) (Bartezzaghi et al. 2015; Yang et al. 2014) and the solve phase (step 4) (Georgescu et al. 2013).

Partial Acceleration

Any one level of the FEA can be accelerated individually. Early work on GPU acceleration focused on porting only portions of the analysis to the GPU to limit significant changes to the programming structure of existing FEA code. As one of the most expensive steps, typically, the analysis is only partially accelerated by moving the solver from the CPU (host) to the GPU (device) (Lu and Guan 2017; Tian et al. 2015); see Fig. 5(b). However, synchronization between the CPU and GPU must then occur every iteration, becoming a bottleneck. In partially accelerated FEA, the CPU must frequently copy a large amount of information needed for the solver to the GPU (Gigabytes of data) and receive results (also in Gigabytes)

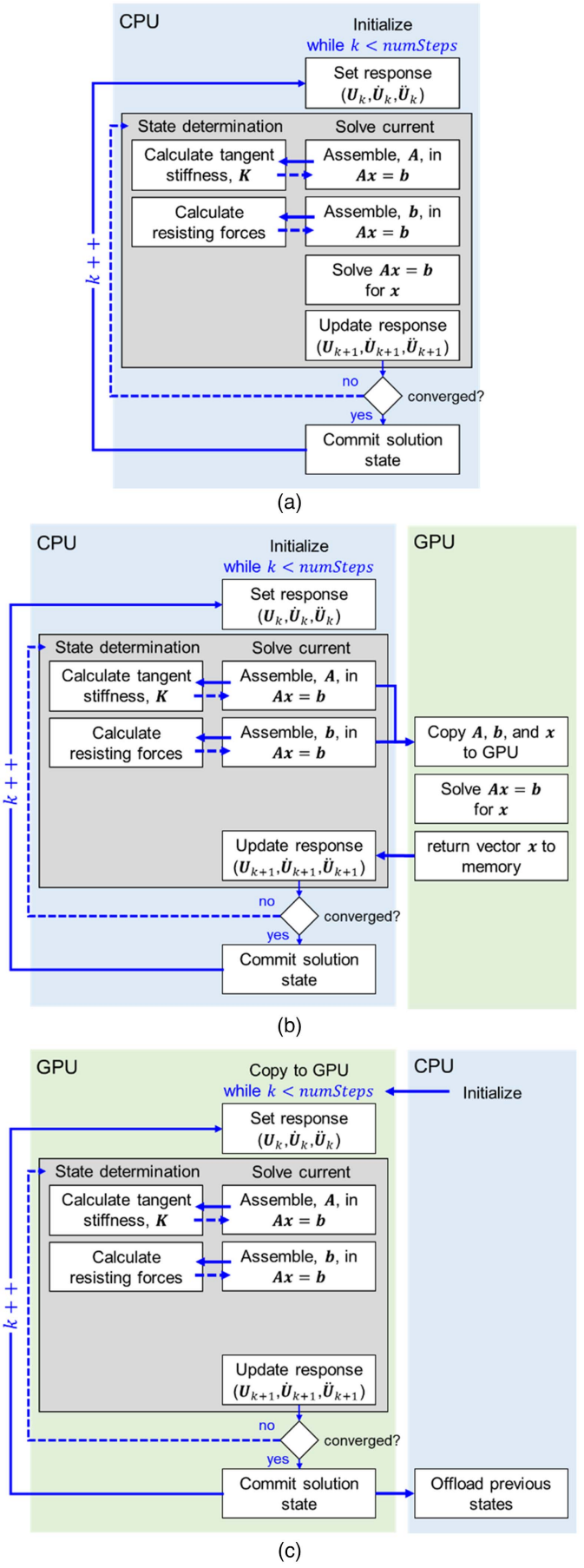

Fig. 5. CPU- versus GPU-based FEA: (a) CPU-only; (b) partial GPU acceleration; and (c) full GPU acceleration.

from the GPU every iteration. This bidirectional data copying via global memory results in time delays and can limit overall speedups, as observed by Lu and Guan (2017). This host-device solver approach is used in Ansys Mechanical (Posey and Courteille 2012), Abaqus (Crivelli and Dunbar 2012), and LS-Dyna (Gohner 2012) with speedups of only 1–3x (Georgescu et al. 2013; NVIDIA 2015).

Full Acceleration

Since many existing FEA codes were built to run sequentially on CPUs, the piece-meal approach used in partial acceleration can only accelerate an FEA so far. Some have suggested that a complete rearchitecture of the standard FEA may be needed to reap the full benefits of GPUs (Bartezzaghi et al. 2015; Garland and Kirk 2010; Knepley and Terrel 2011), see Fig. 5(c), where:

1. Initial preprocessing jobs, like memory allocation, are done from the CPU host.
2. Input data are then copied from the host to the GPU device.
3. Kernels associated with the time-loop for FEA are executed on the GPU.
4. The resulting data are then copied from the device back to the host.

In this case, pre/postprocessing is only done once on the CPU, and the full analysis is performed by the GPU. In the case that there is not enough GPU memory, asynchronous memory transfer can be used to move on to the next time step while offloading results from previous steps to the CPU to limit the idle time and hide the time taken for the data transfer operation (Cai et al. 2015).

GPU Implementation

For full acceleration, the domain of the model can be decomposed according to the parallel portions of the code. Generally, in FEA individual element and nodal update computations are "embarrassingly parallel" and can be performed simultaneously. As calculations of the force contributions and updating the motions for the elements and nodes are nearly independent of their neighbors, two loops can be executed to compute the element state and node motion simultaneously on the GPU. Common parallel execution strategies include: one-thread-per-element, one-thread-per-node, or one-thread-per-dof. In some cases, a one-thread-per-integration point has also been utilized (Komatitsch et al. 2009, 2010; Mihaila et al. 2014).

A complete discussion of per-element, per-node, and per-dof strategies is described in Bartezzaghi et al. (2015) for full computation on the GPU (along with careful memory management). For example, the loops in a serial analysis scheme can correspond to separate kernels for the elements and nodes (Joldes et al. 2010; Taylor et al. 2008). However, execution on the GPU should also be conducted with as few kernels as possible to minimize the number of kernels (which have small software/hardware overhead upon invocation) and information transfer from the host to the device, which is slow. Some have implemented the entire time-stepping algorithm as a single kernel, thereby avoiding multiple kernels. For example, each element can keep a local copy storing information about its nodes and can perform time integration on its own nodes, resulting in an element-wise single kernel (Bartezzaghi et al. 2015). This strategy results in some redundant computations but less memory access as communication between elements is only needed during the assembly process.

Importantly, although considerable efforts have focused on optimized domain decomposition for CPUs (Baugh and Sharma 1994; El-Sayad and Hsiung 1990; Farhat et al. 1987; Foley and Vinnakota 1994; Roa et al. 1994; Synn and Fulton 1995; Zhang and Lui 1991), multicore CPU codes are based on coarse-grained message-passing architectures. In contrast, finer-grained parallelism (Che et al. 2008) is needed for GPUs. For example, the virtual function calls used to handle class hierarchies and element types for finite-element programs like OpenSees (McKenna et al. 2010) often represent branch structures to cover hundreds of classes and could represent inefficient kernels with diverging branch structures on GPUs, potentially lowering performance (Yang et al. 2014). The data on the GPU memory must also be reorganized to obtain optimum memory access performance.

Accelerating the Levels of FEA

Not all prior work on accelerating analyses is suitable for the nonlinear dynamic analysis of civil structures using the FEA method. For example, physics-based gaming engines on GPUs use simplifications that are often not accurate enough for structural analysis (NVIDIA 2012). On the other hand, scientific approaches often accelerate models with a very fine, repetitive mesh or particles with little communication, e.g., discontinuous Galerkin (Klockner et al. 2009), lattice-Boltzmann methods (Kuznik et al. 2010; Zhou et al. 2012), smoothed particle hydrodynamics (SPH) (Dalrymple et al. 2010; Hérault et al. 2010).

Based on the literature, the following sections highlight the challenges associated with each level of the FEA of civil structures in terms of: (1) *state determination*, (2) *assembly* (3) *solver*, and (4) *time-stepping integration scheme*; see Table 2. Each of these sections can be conducted partially or as part of a full acceleration strategy.

State Determination

In FEA, numerical integration is used to determine the element state (e.g., restoring forces, stiffness) as a function of nonlinear constitutive response. Each element has its own state data, such as internal stresses, internal strains, material properties, and inelastic status, which is not shared with other elements. Thus, state

Table 2. Challenges in FEA acceleration of civil structures

Algorithm	Challenge	Example references
State determination	heterogeneous element formulations; conditional branching for nonlinear constitutive material models	Kusakabe et al. (2021) and Yang et al. (2014)
Assembly	Coupled equations of motion; unstructured mesh	Cecka et al. (2011) and Komatitsch et al. (2009)
Matrix solver	Direct versus iterative solvers; selection of preconditioner	Bolz et al. (2003), Georgescu et al. (2013), and Li and Saad (2010)
Time integration	Stability for "stiff" structural problems	Bartezzaghi et al. (2015), Courtecuisse et al. (2010), Joldes et al. (2010), Mafi (2013), and Taylor et al. (2008)

determination is easily parallelized as it involves a loop over the elements and strain/stress calculations.

Importantly, unlike CPUs, the threads performing assembly operate on each element simultaneously. A finer mesh does not necessarily result in increased run time on GPUs, provided the GPU has enough memory for the increased resolution of the model. Simultaneous assembly has significant implications for choosing element formulations suited to refined meshes (e.g., displacement- versus force-based elements). Displacement-based formulations, which do not iterate, are more easily incorporated in GPU codes since stresses are evaluated directly from strains. Reduced time also makes shell (Bartezzaghi et al. 2015; Cai et al. 2015; Martínez-Frutos et al. 2015; Yang et al. 2014) and solid [e.g., tetrahedral (Courtecuisse et al. 2010; Johnsen et al. 2015; Joldes et al. 2010; Kusakabe et al. 2019, 2021; Mafi 2013; Taylor et al. 2008) and hexahedral (Johnsen et al. 2015; Joldes et al. 2010; Komatitsch et al. 2009)] elements more practical.

However, the amount of speedup compared to CPU-based codes depends on the size and resolution of the model; i.e., in terms of the number of dofs. In the literature, higher-order elements (which have many redundant calculations) have shown dramatic speedups (Brown et al. 2022); e.g., greater than 20 times (Komatitsch et al. 2009). However, unless the goal is complete acceleration, the element matrix computations tend to be small compared to the time needed for the solver, particularly for the first and second-order element types commonly used in models of civil structures. Thus, porting this part of the analysis to the GPU may not justify the added effort.

Moreover, GPUs are optimized to perform the same operation repeatedly on huge batches of data, and performance degrades if the heterogeneity of the element formulations and constitutive laws is not considered (Kusakabe et al. 2021). Unlike fluid dynamics, which uses similar elements for every part of the mesh, elements in structural models could be assigned different element formulations, materials, and section types, which require different instructions. Varying degrees of nonlinearity can also result in variable thread execution times and load imbalance between cores handling elastic versus inelastic states.

GPUs also sequentially evaluate both branches of conditional statements and then discard one of the results, which can become costly, as divergent threads in a warp are executed serially. Thus, the calculation of the constitutive law, which often involves conditional branching for different regimes of elastic and inelastic behavior, is unsuitable for GPU computing. Some have suggested separating and reordering elastic from nonlinear elements to model the heterogeneity of soil, where different soil layers (nonliquefiable versus liquefiable) were separated to avoid load imbalance among the processes and threads (Kusakabe et al. 2021). To avoid branching, expressions can be implemented to avoid conditional statements so that every thread in a warp follows the same instructions.

To mitigate challenges in heterogeneous formulations (and, thus, different instructions for different parts of the structure), (Kiran et al. 2019) suggested assigning similar elements to a warp, allowing for different element formulations with varying nonlinearities to be executed at the same time in parallel. Bulk models have also been used to gather elements of the same type (Yang et al. 2014), and each gathered group is then executed in parallel.

Assembly

Contributions of individual elements sharing dofs are assembled and summed to form the global stiffness matrix and righthand side load vector. If the analysis is nonlinear implicit (e.g., using Newton–Raphson iterations), this assembly may need to occur iteratively many times, becoming a costly part of the analysis. Global assembly of the stiffness matrix is often performed in series on CPUs to reduce memory overhead, allowing elements that share dofs access to the same memory locations [e.g., Addto (Markall et al. 2013)]. However, to attain high performance on GPUs, an assembly can be conducted in parallel across all the elements at once, provided enough memory is available on the GPU.

Contributions from each element are computed independently but are summed at the same location in global arrays, where the elements share a dof. As race conditions can arise when adding multiple contributions to the same matrix entry, assembly of elements sharing the same dofs and memory locations can require significant restructuring of the input data on GPUs (Fig. 6). Alternatively, blocking of threads or different mesh constructions can be used to produce a more favorable thread arrangement to achieve memory coalescence. For example, bin numbering schemes, which ensure that neighboring elements are located consecutively in memory, have been proposed for CFD, where boundary elements can be stored consecutively in memory and treated separately from nonboundary elements to reduce thread divergence (Corrigan et al. 2011).

Structured meshes can achieve memory coalescence due to regular memory access patterns; see Fig. 4. However, in general, the unstructured meshes common to civil structures are difficult to pattern for memory coalescence (Bartezzaghi et al. 2015; Courtecuisse et al. 2010; Kusakabe et al. 2021; Taylor et al. 2008). Random global memory access patterns between the coupled portions of the unstructured mesh will be penalized on GPUs (Inoue 2015; Kirk and Hwu 2013). Thus, the irregular topologies (unstructured mesh) in civil structures mean that efficient algorithms based on stencils (Cui et al. 2013; Zhou et al. 2013), e.g., as in finite differences, which can take advantage of cache reuse and optimized prefetch, cannot be applied (Corrigan et al. 2011; Govindaraju and Manocha 2007; Kim 2008).

Methods of restructuring the data to achieve memory coalescence for unstructured meshes have been proposed with varying

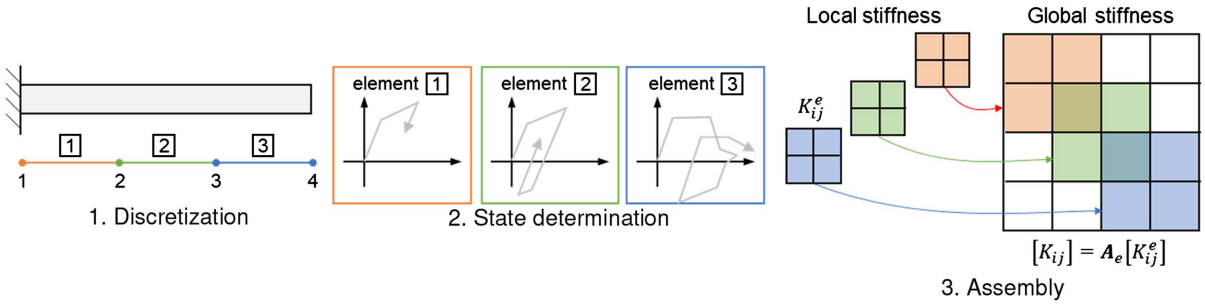

Fig. 6. Assembly process.

J. Struct. Eng., 2023, 149(3): 04022253

speedups (Cecka et al. 2011; Dziekonski et al. 2012; Filipovic et al. 2009; Karatarakis et al. 2014; Kiran et al. 2019; Komatitsch et al. 2009; Luitjens et al. 2012). For example:

- The coloring method (Berger et al. 1982; Farhat and Crivelli 1989; Hughes et al. 1987; Komatitsch et al. 2009) "colors" elements that do not share the same dofs and assembles the different colors sequentially without conflict (Cecka et al. 2011; Dziekonski et al. 2012; Markall et al. 2013). Some rebalancing between colors may be needed (Komatitsch et al. 2009), e.g., per a greedy algorithm (Kiran et al. 2019; Martínez-Frutos et al. 2015).
- Atomic operations are GPU synchronization primitives able to read, modify, and write a value back to device shared memory without the interference of other threads but do so serially to mitigate race conditions if the same memory location is accessed at the same time. Atomic add provided in CUDA can also result in extra instruction overhead and slower run time (Cai et al. 2015). Yamaguchi et al. (2020) found atomic operations to be faster than coloring.

The assembly process can be viewed as a *scatter* operation that assigns each thread to one local stiffness matrix that is then assembled in the global matrix. The scatter approach can involve fewer noncoalesced memory transactions provided there is an efficient use of shared memory; (Mafi and Sirouspour 2014) proposed two levels of atomic add over elements grouped in thread blocks in shared memory prior to atomic add between thread blocks in global memory. Some have also transformed the parallel scatter operation into a parallel *gather* by duplicating the computations for each shared dof (Bartezzaghi et al. 2015; Cai et al. 2012; Courtecuisse et al. 2010; Taylor et al. 2008). A temporary buffer can be used to store the result of each redundant computation and then a gather step reading from this buffer can be used to sum the nodes. This double-buffering approach reads and writes inputs and outputs to different buffers that are swapped at the following time step (Bartezzaghi et al. 2015). All the nodes are processed in parallel, resulting in redundant calculations and a temporary buffer, and then summed, reading from this buffer (Courtecuisse et al. 2010).

Matrix-free methods (Liu and Dinavahi 2018; Rumpf and Strzodka 2005) have also been proposed that use highly parallel element-by-element (EbE) computations at the cost of more iterations (Carey et al. 1988).

Ultimately, parallelized assembly requires a complex memory layout on the GPU (Fig. 2) (Ljungkvist 2015; NVIDIA 2013), as communication of element properties between thread blocks using shared memory is fast, but information shared between thread blocks and synchronization with the CPU through global memory is slow (NVIDIA 2007). Thread blocks may also be unable to share the resulting large amounts of data needed for the assembly of very fine meshes.

Matrix Solver

Most structural FEA applications use implicit time-integration schemes, requiring a solution of the resulting linear system of equations, $\boldsymbol{Ax} = \boldsymbol{b}$, that can have sparse characteristics (with Rayleigh damping). Many previous studies (Cevahir et al. 2010; Schenk et al. 2008; Sharma et al. 2013; Tomov et al. 2010) have focused on linear matrix solvers for GPUs, because the linear solution step can be the most computationally demanding step in FEA (Bartezzaghi et al. 2015; Georgescu et al. 2013), and replacing the solver requires little restructuring of the analysis; e.g., as in Ansys Mechanical (Posey and Courteille 2012), Abaqus (Crivelli and Dunbar 2012), and LS-Dyna (Gohner 2012).

Although direct solvers are common in structural engineering applications (Kilic et al. 2004), direct solvers often adopt triangular and elimination decomposition methods, which are difficult to parallelize for GPUs [e.g., OpenSees uses well-developed packages like MUMPS (direct parallel solver on CPUs), SuperLU, UmfPack, and SPOOLES]. In contrast, iterative solvers can be formulated to require only matrix-vector products that are massively parallelizable, with iterations to converge to a solution for $\boldsymbol{x}$ (Brussino and Sonnad 1989), and are often more suitable for GPU acceleration (Fu et al. 2014).

The iterative conjugate-gradient (CG) method (Berry and Plemmons 1987; Hughes et al. 1987; Law 1986) on the GPU is the most popular (Bolz et al. 2003; Buatois et al. 2007; Cevahir et al. 2010; Georgescu and Okuda 2010; Goddeke et al. 2007; Kruger and Westermann 2003; Verschoor and Jalba 2012). The number of iterations to solve for $\boldsymbol{x}$ highly depends on the initial guess, error tolerance, and condition number of $\boldsymbol{A}$, which increases with the problem size and mesh refinement. Preconditioning is often used to replace $\boldsymbol{Ax} = \boldsymbol{b}$ with an equivalent set of equations with a better condition number and less iterations. However, the overhead of using a preconditioner should not cancel out the savings of having fewer iterations.

On GPUs, speedups for CG highly depend on the selection of the: [a] preconditioner (Geveler et al. 2011; Haase et al. 2010; Kraus and Foster 2012; Li and Saad 2010; Neic et al. 2012; Wagner et al. 2012; Wang et al. 2009) and [b] size of the model; i.e., >500,000 dofs were reported for significant solver speedups in Ansys (Beisheim 2010). Many factorization-based preconditioners are model dependent (Kusakabe et al. 2019) and were developed for serial operations on CPUs (e.g., ILU, IC); thus, there is often a tradeoff between preconditioner quality and parallelism. (Georgescu et al. 2013) found that simple brute-force, highly iterative preconditioners, like Jacobi or block-Jacobi, may provide the best GPU performance for structural problems.

To avoid issues of indirect memory access for the assembly and operation of sparse matrices, the matrix need not be explicitly computed in some iterative solvers. For example, EbE preconditioned CG (PCG), sometimes referred to as matrix-free methods, can be used to replace the matrix assembly part of the algorithm with vector assembly, with mixed speedups (Mafi and Sirouspour 2014; Martínez-Frutos et al. 2015; Yamaguchi et al. 2020). However, from a convergence point of view, PCG with assembled global sparse matrices may still be preferable, as compared by Mafi and Sirouspour (2014). Although (Papadrakakis et al. 2011) found that a direct Cholesky solver performed better than a PCG solver for a hybrid CPU-GPU computing environment, the study used outdated GPUs, and the authors noticed faster speedups using PCG with faster GPUs.

Solving sparse systems of equations is a major research topic, involving sophisticated sparse matrix data structures and algorithms. Solvers were extensively compared by Georgescu et al. (2013), with most of the literature using single-precision hardware. Existing solver libraries (NVIDIA 2014) optimized for scientific applications can be leveraged, particularly, AMGCL (Demidov 2019), PETSc (Mills et al. 2021), and AMGx (Naumov et al. 2015). Libraries for sparse direct solvers for GPUs also exist (Krawezik and Poole 2009; Lacoste et al. 2012; Schenk et al. 2008). Note, the *CuSP* (NVIDIA 2014) solver used by Lu et al. (Lu and Guan 2017) is no longer supported by current GPU versions. Mafi et al. made comparisons with both CUSP and CUSPARSE for compressed sparse row (CSR) sparse matrix-vector multiply (Mafi and Sirouspour 2014).

Time-Integration Scheme

Traditionally, the second-order differential equations of motion governing structural dynamics are discretized in time and solved using implicit time-stepping integration schemes. Often unconditionally stable, implicit methods [e.g., Newmark-beta (Newmark 1959)] need a solver to compute the solution to the system of equations at each time step, which highly depends on the model size and preconditioner, as previously outlined. In contrast, explicit schemes with a lumped (diagonal) mass matrix, enable the equations of motion to become decoupled such that each dof can be solved independently, resulting in calculations that can be easily parallelized on an element- and node-wise basis without iterations. Table 3 summarizes the literature for implicit and explicit time-integration schemes, including notes on the GPU hardware and CPU comparison (serial CPU code).

Real-time GPU-based biomechanics applications often use explicit methods (Johnsen et al. 2015; Joldes et al. 2010; Taylor et al. 2008), but the low stiffness of soft biological tissues means the time step needed for stability, normally restrictive, is relatively large (Taylor et al. 2008). However, common conditionally stable methods, like central difference (Bartezzaghi et al. 2015; Cai et al. 2015; Joldes et al. 2010; Taylor et al. 2008) and explicit Newmark (Komatitsch et al. 2009), often impractically restrict the time step size for multi-dof civil structural systems (Dokainish and Subbaraj 1989), even if they do not require iterations to reach convergence.

Explicit time-integration schemes are highly suitable for GPUs, but civil structures are a "stiff" problem (Hairer and Wanner 2012), requiring a small time step to meet stability requirements. Several biomechanics applications have implemented implicit time-integration schemes successfully (Courtecuisse et al. 2010; Mafi 2013; Mafi and Sirouspour 2014). However, thread divergence may occur more often for implicit integration schemes, which tend to be programmed with more complex branching compared to explicit methods (Stone and Davis 2013). Some have suggested semiimplicit schemes that do not require Newton iterations (instead, they solve a sequence of linear systems) may be better suited to SIMT acceleration (no thread divergence); e.g., semi-implicit and implicit Runge-Kutta methods have been used in GPU acceleration of "stiff" chemical reaction applications (Curtis et al. 2017).

Demonstration Study

To benchmark potential speedups, a simple platform was built to fully GPU-accelerate a linear-elastic dynamic analysis that ported the assembly, solver, and update tasks to the GPU device. The CPU instructs the GPU on how to perform the computations (NVIDIA 2007), but the output was only returned to the CPU upon completion of the analysis. Importantly, once the analysis was sent to the GPU, data never left the GPU until the analysis was complete.

Generic structural models were generated using elastic beam-column elements in three ways: (1) m randomly connected elements within a cube domain containing n randomly positioned nodes, (2) a 1D mesh with N nodes numbered sequentially from 1 to N at the end (i.e., a banded matrix), and (3) a regular 3D frame structure defined by the number of stories, bays in each direction, and nodes per member. The first two model types were used when constructing the GPU-based code. Performance was then defined based on the number of dofs in the model for the different types of mash structure for the third model type, which best represents a regular structure.

A new CPU-only analysis was written to be one-to-one with GPU-based code for this comparison. The simulation was executed on Oregon State University's NVIDIA DGX-2 cluster with NVIDIA Tesla V100 GPUs using CUDA version 10.0 and compared to those executed on a single-core Intel Xeon CPU at 3.4 GHz. All calculations were performed in double precision.

Pseudo-Code

In the GPU-accelerated code, synchronization only occurs at the model input and final output stage. The overall code is set up with the following pseudo-code:

1. Define inputs and set configuration files on the CPU.
2. *domain().load():* Load model from the CPU to the GPU (e.g., load nodal data, element connectivity, element data (elasticity), etc.).
3. *domain().changed():* Create data structures on GPU (e.g., nodal dofs, nodal displacements, velocities, accelerations, mass, etc.).
4. *integrator().new_step(dt):* Setup the Newmark constants on the GPU.
5. *assembler().assemble():* Assemble global matrix (left-hand side) and vector (righthand side) on the GPU.

Table 3. Time-integration schemes

Reference	Method	Integration scheme	Solver	Field	Precision	Estimated speedup	CPU[a] comparison
Bartezzaghi et al. (2015)	Explicit	Central difference	—	Solid mechanics; structural dynamics	DP	>40x	serial CPU[b]
Cai et al. (2015)		Central difference	—	Sheet metal; crashworthiness	DP	7-22x	serial CPU
Taylor et al. (2008)		Central difference*	—	Biomechanics	SP	10-17x[c]	serial CPU
Joldes et al. (2010)		Central difference*	—	Biomechanics	SP	>20x	serial CPU
Komatitsch et al. (2009)		Explicit Newmark	—	Seismic modeling	SP	25x	serial CPU
Mafi and Sirouspour (2014)	Implicit	Implicit Newmark	CG versus EbE CG	Biomechanics	SP	10x	serial CPU
Yamaguchi et al. (2020)		Implicit Newmark	EbE CG	Seismic modeling	mixed-precision with FP21	11–16x[d]	serial CPU
Courtecuisse et al. (2010)		Backward Euler	EbE CG	Biomechanics	DP	15–35x	serial CPU

Note: *Total Lagrangian explicit dynamics.
[a]Speedups are difficult to compare across published works due to different CPU and GPU hardware and model and resolution (in terms of dofs).
[b]Compared to commercial FEA codes.
[c]GPU acceleration with graphics primitives. All other GPU acceleration with CUDA.
[d]GPU acceleration with OpenACC.

6. *solver().solve():* Solves the linear system of equations using the AMGX iterative solver.
7. *domain().update():* Update nodal displacements and calculate velocities and accelerations based on Newmark constants. Commit the final state.
8. Pass committed state back to CPU.

Step 3 sets up the data structures defining the memory access on the GPU. Memory is pre-allocated for all entries into global memory. Extra memory is also allocated for duplicate element entries sharing the same node. All nonzero entries are gathered based on the nodes (and their dofs) for each element. All element entries are then sorted based on the first element node, ensuring that duplicate entries arising from entries sharing the same node are adjacent to each other in global memory. In preparation to solve $\boldsymbol{Ax} = \boldsymbol{b}$, the CSR format allocates memory for pointers to the nonzero matrix and vector values.

Assembly in Step 5 is then conducted by mapping element contributions from GPU local to global memory based on the resulting equation numbering and: (1) considering all nonzero entries, (2) gathering nodes with the same dofs, (3) leaving additional memory space for shared nodes, (4) assembling all contributions from the elements to the global stiffness matrix in parallel at once, and (5) summing shared dof contributions at overlapping equation numbers where multiple elements were connected to the same node.

Speedups

The speedup for a single analysis step was compared between the GPU- and CPU-based codes in Fig. 7 based on assembly, solve, and update levels of the linear-elastic analysis. Total run times are shown separately for the GPU- and equivalent CPU-based analyses in Figs. 7(a and b) and broken down by timings for the assembly, update, and solve stages. Fig. 7(c) shows the overall speedups, defined as the ratio of the GPU computational time divided by the CPU computational time, with respect to the number of dofs. The horizontal line at 10^0 indicates equivalent CPU and GPU run times.

For a single time step and 10^6 dofs, observed speedups were approximately 115 times that of an equivalent CPU-driven code. Time for the CPU code increased with the number of dofs. In contrast, time on the GPUs was near-constant up to approximately 10^4 dofs when the GPU had to run more than one pass through the available cores. Notably, GPU computing is massively parallel, and the GPU code could operate simultaneously on each node and element in the model at once. As such, the computational time was nearly independent of the number of dofs.

A CG solver with a block-Jacobi preconditioner using NVIDIA's geometry-informed algebraic multigrid [AMGx (Naumov et al. 2015)] package was used to solve the linear system of equations. Despite the speedups observed for the assembly and update levels, the solver on the GPU remained a bottleneck compared to the CPU depending on the number of dofs; as shown in Fig. 7(c).

The intent of this study was to demonstrate the potential of GPU-accelerated structural applications. However, the aforementioned challenges still need to be addressed to capitalize on GPU-driven speedups for nonlinear structural systems. Although promising and prevalent in other fields, e.g., seismic wave propagation, 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 exploring GPU acceleration including nonlinear response with heterogeneous elements and inelastic material formulations.

Conclusions

To guide future endeavors in accelerating FEA using GPUs, this state-of-the-art review presents the existing literature on GPU-accelerated structural and solid mechanics applications with a focus on structures subjected to seismic loading. A demonstration study was used to assess the feasibility of a fully GPU-accelerated analysis. The demonstration study suggests that GPU acceleration is promising for linear-elastic analyses. Importantly, finer discretization—often associated with increased accuracy—did not necessarily result in increased run time on GPUs for the assembly and update steps of FEA, because the GPU can operate on each dof at once.

However, GPUs are not a panacea for all scientific problems (Owens et al. 2005); i.e., not all applications are well suited to GPU acceleration, particularly if the system behaves in the nonlinear range. Although promising, other solvers still need to be assessed, comparisons were not made to multicore CPUs, and nonlinear behavior was not implemented. A baseline for effective comparison of multicore CPU to GPU computing would give a more effective comparison of speedups; i.e., speedups against unicore CPUs may look more impressive than multicore CPUs. Based on the literature, specific challenges for the GPU acceleration of the nonlinear FEA include:

- The GPU acceleration of civil structures under dynamic loads poses unique problems compared to other GPU-accelerated applications; e.g., in implementing extreme nonlinearities associated with inelastic constitutive laws, as for seismic loading. The seismic response of civil structures is composed of varied material and element formulations and patterns of inelastic and nonlinear behavior that is not uniform across the entire structure, which can be challenging for the massive single-instruction parallelism desirable for GPUs.
- Structural models often have complex meshes, which result in irregular, unstructured sparse matrices hard to organize for GPUs; i.e., the topology is not regular and cannot be described

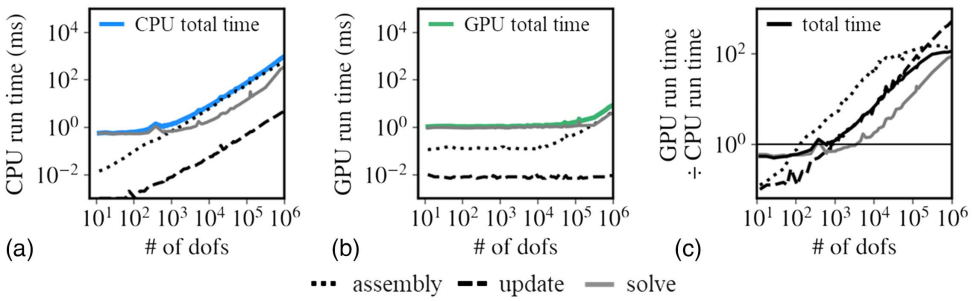

Fig. 7. Speedups of demonstration study.

J. Struct. Eng., 2023, 149(3): 04022253

by simply (i, j, k) addressing and typically requires a large number of random memory accesses, often resulting in noncoalesced memory access.
- The extent of GPU acceleration depends on the hardware age/performance (Farhat 1990) and the equations being solved. The seismic response of civil structures is in a class of nonlinear dynamics known as inertial problems (Dokainish and Subbaraj 1989), represented by stiff equations dominated by a few low-frequency modes, which pose unique challenges compared to other Lagrangian formulations, e.g., real-time biomechanics (Courtecuisse et al. 2010; Johnsen et al. 2015; Joldes et al. 2010; Taylor et al. 2008).

Despite these challenges, it is expected that this review will facilitate the use of HPC in structural analysis applications. GPUs are only part of a larger paradigm shift toward customized hardware offering finer-grained parallelism for scientific applications. With advancements in exascale deployment, GPUs are only the first of many realizations of emerging hardware that structural engineers can leverage to enhance understanding of multihazard phenomena and the design of more resilient and sustainable urban environments.

Intractable computational times are a significant obstacle to promoting advanced collapse-prevention and life-saving structural design methods; e.g., high degrees of nonlinearity may require refinement from beam-column elements to finer shell or solid elements. Accelerating analyses would also be a step toward establishing physics-based, end-to-end models capable of spanning scales (O'Rourke 2010); e.g., from the molecular-to-component-to-structure-to-urban scales, which is inhibited by computational time (Ghattas 2011; McCallen et al. 2021a; O'Rourke 2010). As GPUs are often optimized for machine learning, the intersection of GPU-accelerated HPC in FEA and artificial intelligence would promote greater use of machine learning in natural hazards research engineering by housing both the analysis (e.g., for training) and machine learning algorithms on the GPU. Moreover, future concurrent simulation and visualization on the GPU could enable real-time interaction with the data (Tavakkol and Lynett 2017), revolutionizing the approach to advanced analysis and structural design. Ultimately, the ability to conduct higher-fidelity simulations faster would lead to more detailed and accurate models, encouraging more innovative structural systems and devices and enable more computationally-intensive applications in uncertainty propagation, regional-scale modeling, and interaction-type problems, among many other applications.

Data Availability Statement

Some or all data, models, or code that support the findings of this study are available from the corresponding author upon request.

Acknowledgments

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*. Special thanks to Dr. Frank McKenna for providing feedback on the initial drafts of this paper. 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.

References

Alexander, F., et al. 2020. "Exascale applications: Skin in the game." *Philos. Trans. R. Soc. A* 378 (2166): 20190056. https://doi.org/10.1098/rsta.2019.0056.

Amdahl, G. M. 1967. "Validity of the single processor approach to achieving large scale computing capabilities." In *Proc., AFIPS Conf. Proc.*, 483–485. Reston, VA: American Federation of Information Processing Societies.

Anderson, J. A., C. D. Lorenz, and A. Travesset. 2008. "General purpose molecular dynamics simulations fully implemented on graphics processing units." *J. Comput. Phys.* 227 (10): 5342–5359. https://doi.org/10.1016/j.jcp.2008.01.047.

Appleyard, J., and D. Drikakis. 2011. "Higher-order CFD and interface tracking methods on highly-parallel MPI and GPU systems." *Comput. Fluids* 46 (1): 101–105. https://doi.org/10.1016/j.compfluid.2010.10.019.

Bartezzaghi, A., M. Cremonesi, N. Parolini, and U. Perego. 2015. "An explicit dynamics GPU structural solver for thin shell finite elements." *Comput. Struct.* 154 (Jul): 29–40. https://doi.org/10.1016/j.compstruc.2015.03.005.

Bathe, K. J. 1996. *Finite element procedures*. 1st ed. Upper Saddle River, NJ: Prentice Hall.

Baugh, J. W., Jr., and S. K. Sharma. 1994. "Evaluation of distributed finite element algorithms on a workstation network." *Eng. Comput.* 10 (1): 45–62. https://doi.org/10.1007/BF01206539.

Beckingsale, D. A., J. Burmark, R. Hornung, H. Jones, W. Killian, A. J. Kunen, O. Pearce, P. Robinson, B. S. Ryujin, and T. R. W. Scogland. 2019. "RAJA: Portable performance for large-scale scientific applications." In *Proc., 2019 IEEE/ACM Int. Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. New York: IEEE. https://doi.org/10.1109/P3HPC49587.2019.00012.

Beisheim, J. 2010. "Speed up simulations with a GPU." In *ANSYS Advantage*, 6–8. Canonsburg, PA: ANSYS.

Berger, P., P. Brouaye, and J. C. Syre. 1982. "A mesh coloring method for efficient MIMD processing in finite element problems." In *Proc., of the Int. Conf. on Parallel Processing, ICPP'82*, 41–46. Washington, DC: IEEE Computer Society.

Berry, M. W., and R. J. Plemmons. 1987. "Algorithms and experiments for structural mechanics on high-performance architectures." *Comput. Methods Appl. Mech. Eng.* 64 (1–3): 487–507. https://doi.org/10.1016/0045-7825(87)90052-1.

Bolz, J., I. Farmer, E. Grinspun, and P. Schrooder. 2003. "Sparse matrix solvers on the GPU: Conjugate gradients and multigrid." In *Proc., ACM SIGGRAPH 2003 Papers*, 917–924. New York: Association for Computing Machinery.

Borkar, S., P. Dubey, K. Kahn, D. Kuck, H. Mulder, S. Pawlowski, and J. Rattner. 2005. "Platform 2015: Intel processor and platform evolution for the next decade." In *Intel White Paper*, 30–36. Santa Clara, CA: Intel.

Brandvik, T., and G. Pullan. 2007. "Acceleration of a two-dimensional Euler flow solver using commodity graphics hardware." *Proc. Inst. Mech. Eng., Part C: J. Mech. Eng. Sci.* 221 (12): 1745–1748. https://doi.org/10.1243/09544062JMES813FT.

Brown, J., V. Barra, N. Beams, L. Ghaffari, M. Knepley, W. Moses, R. Shakeri, K. Stengel, J. Thompson, and J. Zhang. 2022. "Performance portable solid mechanics via matrix-free p-multigrid." Preprint, submitted April 4, 2022. https://arxiv.org/abs/2204.01722.

Brussino, G., and V. Sonnad. 1989. "A comparison of direct and preconditioned iterative techniques for sparse, unsymmetric systems of linear equations." *Int. J. Numer. Methods Eng.* 28 (4): 801–815. https://doi.org/10.1002/nme.1620280406.

Buatois, L., G. Caumon, and B. Levy. 2007. "Concurrent number cruncher: An efficient sparse linear solver on the GPU." In *Proc., Int. Conf. on High Performance Computing and Communications*, 358–371. Berlin: Springer.

Byna, S., M. S. Breitenfeld, B. Dong, Q. Koziol, E. Pourmal, D. Robinson, J. Soumagne, H. Tang, V. Vishwanath, and R. Warren. 2020. "ExaHDF5: Delivering efficient parallel I/O on exascale computing systems." *J. Comput. Sci. Technol.* 35 (1): 145–160. https://doi.org/10.1007/s11390-020-9822-9.

Cai, Y., G. Li, H. Wang, G. Zheng, and S. Lin. 2012. "Development of parallel explicit finite element sheet forming simulation system based on GPU architecture." *Adv. Eng. Software* 45 (1): 370–379. https://doi.org/10.1016/j.advengsoft.2011.10.014.

Cai, Y., G. Wang, G. Li, and H. Wang. 2015. "A high performance crashworthiness simulation system based on GPU." *Adv. Eng. Software* 86 (Aug): 29–38. https://doi.org/10.1016/j.advengsoft.2015.04.003.

Carey, G. F., E. Barragy, R. McLay, and M. Sharma. 1988. "Element-by-element vector and parallel computations." *Commun. Appl. Numer. Methods* 4 (3): 299–307. https://doi.org/10.1002/cnm.1630040303.

Cecka, C., A. Lew, and E. Darve. 2011. "Assembly of finite element methods on graphics processors." *Int. J. Numer. Methods Eng.* 85 (5): 640–669. https://doi.org/10.1002/nme.2989.

Cevahir, A., A. Nukada, and S. Matsuoka. 2010. "High performance conjugate gradient solver on multi-GPU clusters using hypergraph partitioning." *Comput. Sci.-Res. Dev.* 25 (1–2): 83–91. https://doi.org/10.1007/s00450-010-0112-6.

Che, S., M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron. 2008. "A performance study of general-purpose applications on graphics processors using CUDA." *J. Parallel Distrib. Comput.* 68 (10): 1370–1380. https://doi.org/10.1016/j.jpdc.2008.05.014.

Chen, W., Y. Zhu, F. Cui, L. Liu, Z. Sun, J. Chen, and Y. Li. 2016. "GPU-accelerated molecular dynamics simulation to study liquid crystal phase transition using coarse-grained Gay-Berne anisotropic potential." *PloS one* 11 (3): e0151704. https://doi.org/10.1371/journal.pone.0151704.

Corrigan, A., F. F. Camelli, R. Löhner, and J. Wallin. 2011. "Running unstructured grid-based CFD solvers on modern graphics hardware." *Int. J. Numer. Methods Fluids* 66 (2): 221–229. https://doi.org/10.1002/fld.2254.

Courtecuisse, H., H. Jung, J. Allard, C. Duriez, D. Y. Lee, and S. Cotin. 2010. "GPU-based real-time soft tissue deformation with cutting and haptic feedback." *Prog. Biophys. Mol. Biol.* 103 (2–3): 159–168. https://doi.org/10.1016/j.pbiomolbio.2010.09.016.

Crivelli, L., and M. Dunbar. 2012. "Evolving use of GPU for Dassault Systemes simulation products." In *Proc., GPU Technology Conf. (GTC 2012*. San Jose, CA: NVIDIA.

Cui, Y., et al. 2013. "Physics-based seismic hazard analysis on petascale heterogeneous supercomputers." In *Proc., SC'13: Proc. of the Int. Conf. on High Performance Computing, Networking, Storage and Analysis*. New York: IEEE.

Curtis, N. J., K. E. Niemeyer, and C.-J. Sung. 2017. "An investigation of GPU-based stiff chemical kinetics integration methods." *Combust. Flame* 179 (May): 312–324. https://doi.org/10.1016/j.combustflame.2017.02.005.

Dalrymple, R. A., A. Hérault, G. Bilotta, and R. J. Farahani. 2010. "GPU-accelerated SPH model for water waves and free surface flows." In *Proc., of the Coastal Engineering Conf.* Reston, VA: ASCE.

Demidov, D. 2019. "AMGCL: An efficient, flexible, and extensible algebraic multigrid implementation." *Lobachevskii J. Math.* 40: 535–546. https://doi.org/10.1134/S1995080219050056.

Dokainish, M. A., and K. Subbaraj. 1989. "A survey of direct time-integration methods in computational structural dynamics—I. Explicit methods." *Comput. Struct.* 32 (6): 1371–1386. https://doi.org/10.1016/0045-7949(89)90314-3.

Dziekonski, A., P. Sypek, A. Lamecki, and M. Mrozowski. 2012. "Finite element matrix generation on a GPU." *Prog. Electromagn. Res.* 128 (Apr): 249–265. https://doi.org/10.2528/PIER12040301.

Elgamal, A., L. Yan, Z. Yang, and J. P. Conte. 2008. "Three-dimensional seismic response of Humboldt Bay bridge-foundation-ground system." *J. Struct. Eng.* 134 (7): 1165–1176. https://doi.org/10.1061/(ASCE)0733-9445(2008)134:7(1165).

El-Sayad, M. E. M., and C.-K. Hsiung. 1990. "Parallel finite element computation with separate substructures." *Comput. Struct.* 36 (2): 261–265. https://doi.org/10.1016/0045-7949(90)90125-L.

Farhat, C. 1990. "Which parallel finite element algorithm for which architecture and which problem?" *Eng. Comput.* 7 (3): 186–195. https://doi.org/10.1108/eb023805.

Farhat, C., and L. Crivelli. 1989. "A general approach to nonlinear FE computations on shared-memory multiprocessors." *Comput. Methods Appl. Mech. Eng.* 72 (2): 153–171. https://doi.org/10.1016/0045-7825(89)90157-6.

Farhat, C., E. Wilson, and G. Powell. 1987. "Solution of finite element systems on concurrent processing computers." *Eng. Comput.* 2 (3): 157–165. https://doi.org/10.1007/BF01201263.

FEMA. 1996. *Performance based seismic design of buildings: An action plan for future studies*. FEMA 283. Washington, DC: FEMA.

Filipovic, J., I. Peterlik, and J. Fousek. 2009. "GPU acceleration of equations assembly in finite elements method—Preliminary results." In *Proc., Symp. on Application Accelerators in HPC (SAAHPC)*. Urbana, IL: US National Center for Supercomputing Applications.

Foley, C. M., and S. Vinnakota. 1994. "Parallel processing in the elastic nonlinear analysis of high-rise frameworks." *Comput. Struct.* 52 (6): 1169–1179. https://doi.org/10.1016/0045-7949(94)90183-X.

Fu, Z., T. J. Lewis, R. M. Kirby, and R. T. Whitaker. 2014. "Architecting the finite element method pipeline for the GPU." *J. Comput. Appl. Math.* 257 (Feb): 195–211. https://doi.org/10.1016/j.cam.2013.09.001.

Garland, M., and D. B. Kirk. 2010. "Understanding throughput-oriented architectures." *Commun. ACM* 53 (11): 58–66. https://doi.org/10.1145/1839676.1839694.

Georgescu, S., P. Chow, and H. Okuda. 2013. "GPU Acceleration for FEM-based structural analysis." *Arch. Comput. Methods Eng.* 20 (2): 111–121. https://doi.org/10.1007/s11831-013-9082-8.

Georgescu, S., and H. Okuda. 2010. "Conjugate gradients on multiple GPUs." *Int. J. Numer. Methods Fluids* 64 (10–12): 1254–1273. https://doi.org/10.1002/fld.2462.

Geveler, M., D. Ribbrock, D. Gdeke, P. Zajac, and S. Turek. 2011. "Efficient finite element geometric multigrid solvers for unstructured grids on GPUs." In *Proc., 2nd Int. Conf. on Parallel, Distributed, Grid and Cloud Computing for Engineering (PARENG 2011)*. Corsica, France: Civil-Comp.

Ghattas, O. 2011. "Uncertainty quantification and exascale computing: Opportunities and challenges for earthquake engineering." In *Proc., Grand Challenges in Earthquake Engineering Research: A Community Workshop Report*, 74–80. Washington DC: National Research Council of the National Academies.

Gimenez, J. M., D. E. Ramajo, S. Márquez Damián, N. M. Nigro, and S. R. Idelsohn. 2017. "An assessment of the potential of PFEM-2 for solving long real-time industrial applications." *Comput. Part. Mech.* 4 (3): 251–267. https://doi.org/10.1007/s40571-016-0135-2.

Goddeke, D., R. Strzodka, and S. Turek. 2007. "Performance and accuracy of hardware-oriented native-, emulated- and mixed-precision solvers in FEM simulations." *Int. J. Parallel Emergent Distrib. Syst.* 22 (4): 221–256. https://doi.org/10.1080/17445760601122076.

Gohner, U. 2012. "Usage of GPU in LS-DYNA." In *Proc., LS-DYNA Forum*. Stuttgart, Germany: DYNAmore.

Gorobets, A., and P. Bakhvalov. 2022. "Heterogeneous CPU+GPU parallelization for high-accuracy scale-resolving simulations of compressible turbulent flows on hybrid supercomputers." *Comput. Phys. Commun.* 271 (Feb): 108231. https://doi.org/10.1016/j.cpc.2021.108231.

Govindaraju, N. K., and D. Manocha. 2007. "Cache-efficient numerical algorithms using graphics hardware." *Parallel Comput.* 33 (10–11): 663–684. https://doi.org/10.1016/j.parco.2007.09.006.

Haase, G., M. Liebmann, C. C. Douglas, and G. Plank. 2010. "A parallel algebraic multigrid solver on graphics processing units." In *High performance computing and applications*, 38–47. Berlin: Springer.

Hairer, E., and G. Wanner. 2012. *Solving ordinary differential equations II: Stiff and differential-algebraic problems*. Berlin: Springer-Verlag.

Hajjar, J. F., and J. F. Abel. 1988. "Parallel processing for transient nonlinear structural dynamics of three-dimensional framed structures using domain decomposition." *Comput. Struct.* 30 (6): 1237–1254. https://doi.org/10.1016/0045-7949(88)90189-7.

Hérault, A., G. Bilotta, and R. A. Dalrymple. 2010. "SPH on GPU with CUDA." Supplement, *J. Hydraul. Res.* 48 (S1): 74–79. https://doi.org/10.1080/00221686.2010.9641247.

Hughes, T. J. R. 1987. *The finite element method*. Englewood Cliffs, NJ: Prentice Hall.

Hughes, T. J. R., R. M. Ferencz, and J. O. Hallquist. 1987. "Large-scale vectorized implicit calculations in solid mechanics on a Cray X-MP/48 utilizing EBE preconditioned conjugate gradients." *Comput. Methods*

Appl. Mech. Eng. 61 (2): 215–248. https://doi.org/10.1016/0045-7825(87)90005-3.

Inoue, N. 2015. "Speeding up a finite element computation on GPU." In *Proc., GPU Technology Conf.* Silicon Valley, CA: NVIDIA.

Jeremic, B., and G. Jie. 2008. "Parallel soil–foundation–structure interaction computations." In *Computational structural dynamics and earthquake engineering*. Boca Raton, FL: CRC Press.

Johnsen, S. F., et al. 2015. "NiftySim: A GPU-based nonlinear finite element package for simulation of soft tissue biomechanics." *Int. J. Comput. Assisted Radiol. Surg.* 10 (7): 1077–1095. https://doi.org/10.1007/s11548-014-1118-5.

Joldes, G. R., A. Wittek, and K. Miller. 2010. "Real-time nonlinear finite element computations on GPU—Application to neurosurgical simulation." *Comput. Methods Appl. Mech. Eng.* 199 (49–52): 3305–3314. https://doi.org/10.1016/j.cma.2010.06.037.

Kampolis, I. C., X. S. Trompoukis, V. G. Asouti, and K. C. Giannakoglou. 2010. "CFD-based analysis and two-level aerodynamic optimization on graphics processing units." *Comput. Methods Appl. Mech. Eng.* 199 (9–12): 712–722. https://doi.org/10.1016/j.cma.2009.11.001.

Karatarakis, A., P. Karakitsios, and M. Papadrakakis. 2014. "GPU accelerated computation of the isogeometric analysis stiffness matrix." *Comput. Methods Appl. Mech. Eng.* 269 (Feb): 334–355. https://doi.org/10.1016/j.cma.2013.11.008.

Kilic, S. A., F. Saied, and A. Sameh. 2004. "Efficient iterative solvers for structural dynamics problems." *Comput. Struct.* 82 (28): 2363–2375. https://doi.org/10.1016/j.compstruc.2004.06.001.

Kim, T. 2008. "Hardware-aware analysis and optimization of 'Stable fluids'." In *Proc., of the ACM Symp. on Interactive 3D Graphics and Games*. New York: Association for Computing Machinery.

Kiran, U., D. Sharma, and S. S. Gautam. 2019. "GPU-warp based finite element matrices generation and assembly using coloring method." *J. Comput. Des. Eng.* 6 (4): 705–718. https://doi.org/10.1016/j.jcde.2018.11.001.

Kirk, D. B., and W.-M. W. Hwu. 2013. *Programming massively parallel processors: A hands-on approach*. Waltham, MA: Morgan Kaufmann.

Klockner, A., T. Warburton, J. Bridge, and J. S. Hesthaven. 2009. "Nodal discontinuous Galerkin methods on graphics processors." *J. Comput. Phys.* 228 (21): 7863–7882. https://doi.org/10.1016/j.jcp.2009.06.041.

Knepley, M. G., and A. R. Terrel. 2011. *Finite element integration on GPUs*. Austin, TX: Texas Advanced Computing Center.

Komatitsch, D., G. Erlebacher, D. Göddeke, and D. Michéa. 2010. "High-order finite-element seismic wave propagation modeling with MPI on a large GPU cluster." *J. Comput. Phys.* 229 (20): 7692–7714. https://doi.org/10.1016/j.jcp.2010.06.024.

Komatitsch, D., D. Michéa, and G. Erlebacher. 2009. "Porting a high-order finite-element earthquake modeling application to NVIDIA graphics cards using CUDA." *J. Parallel Distrib. Comput.* 69 (5): 451–460. https://doi.org/10.1016/j.jpdc.2009.01.006.

Kothe, D., L. Diachin, A. Siegel, and E. Draeger. 2019. *Application development update*. Washington, DC: Exascale Computing Project.

Kraus, J., and M. Foster. 2012. "Efficient AMG on heterogeneous systems." In Vol. 7174 of *Facing the multicore—Challenge II, lecture notes in computer science*, edited by R. Keller, D. Kramer, and J. P. Weiss, 133–146. Berlin: Springer.

Krawezik, G., and G. Poole. 2009. "Accelerating the ANSYS direct sparse solver with GPUs." In *Proc., 2009 Symp. on Application Accelerators in High Performance Computing (SAAHPC'09)*. Urbana, IL: US National Center for Supercomputing Applications.

Kruger, J., and R. Westermann. 2003. "Linear algebra operators for GPU implementation of numerical algorithms." *ACM Trans. Graphics* 22 (3): 908–916. https://doi.org/10.1145/882262.882363.

Kumar, S., and H. Adeli. 1995. "Distributed finite-element analysis on network of workstations—Implementation and application." *J. Struct. Eng.* 121 (10): 1456–1462. https://doi.org/10.1061/(ASCE)0733-9445(1995)121:10(1456).

Kusakabe, R., K. Fujita, T. Ichimura, M. Hori, and L. Wijerathne. 2019. "A fast 3D finite-element solver for large-scale seismic soil liquefaction analysis." In Vol. 11537 of *Proc., Int. Conf. on Computational Science, Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 349–362. Cham, Switzerland: Springer.

Kusakabe, R., K. Fujita, T. Ichimura, T. Yamaguchi, M. Hori, and L. Wijerathne. 2021. "Development of regional simulation of seismic ground-motion and induced liquefaction enhanced by GPU computing." *Earthquake Eng. Struct. Dyn.* 50 (1): 197–213. https://doi.org/10.1002/eqe.3369.

Kuznik, F., C. Obrecht, G. Rusaouen, and J.-J. Roux. 2010. "LBM based flow simulation using GPU computing processor." *Comput. Math. Appl.* 59 (7): 2380–2392. https://doi.org/10.1016/j.camwa.2009.08.052.

Lacoste, X., P. Ramet, M. Faverge, Y. Ichitaro, and J. Dongarra. 2012. *Sparse direct solvers with accelerators over DAG runtimes*. Research Rep. RR-7972. Talence, France: INRIA Bordeaux.

Law, K. H. 1986. "A parallel finite element solution method." *Comput. Struct.* 23 (6): 845–858. https://doi.org/10.1016/0045-7949(86)90254-3.

Leung, A., N. Vasilache, B. Meister, M. Baskaran, D. Wohlford, C. Bastoul, and R. Lethin. 2010. "A mapping path for multi-GPGPU accelerated computers from a portable high level programming abstraction." In *Proc., 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, 51–61. New York: Association for Computing Machinery.

Li, R., and Y. Saad. 2010. "*GPU-accelerated preconditioned iterative linear solvers*." Technical Rep. Minneapolis: Univ. of Minnesota.

Lindstrom, P. 2014. "Fixed-rate compressed floating-point arrays." *IEEE Trans. Visual Comput. Graphics* 20 (12): 2674–2683. https://doi.org/10.1109/TVCG.2014.2346458.

Liu, P., and V. Dinavahi. 2018. "Matrix-free nodal domain decomposition with relaxation for massively parallel finite-element computation of EM apparatus." *IEEE Trans. Magn.* 54 (9): 1–7. https://doi.org/10.1109/TMAG.2018.2848622.

Ljungkvist, K. 2015. "Techniques for finite element methods on modern processors." Ph.D. dissertation, Dept. of Information Technology, Uppsala Univ.

Lu, X., and H. Guan. 2017. *Earthquake disaster simulation of civil infrastructures: From tall buildings to urban areas*. Beijing: Science Press.

Luitjens, J., A. Williams, and M. Heroux. 2012. "Optimizing MiniFE an implicit nite element application on GPUs." In *Proc., GPU Technology Conf. (GTC 2012)*. Santa Clara, CA: NVIDIA.

Mackerle, J. 1996. "Implementing finite element methods on supercomputers, workstations and PCs: A bibliography (1985-1995)." *Eng. Comput.* 13 (1): 33–85. https://doi.org/10.1108/02644409610110985.

Mackerle, J. 2003. "FEM and BEM parallel processing: Theory and applications—A bibliography (1996-2002)." *Eng. Comput.* 20 (4): 436–484. https://doi.org/10.1108/02644400310476333.

Mackerle, J. 2004. "Object-oriented programming in FEM and BEM: A bibliography (1990-2003)." *Adv. Eng. Software* 35 (6): 325–336. https://doi.org/10.1016/j.advengsoft.2004.04.006.

Mafi, R. 2013. *GPU-based parallel computing for nonlinear finite element deformation analysis*. Hamilton, ON, Canada: McMaster Univ.

Mafi, R., and S. Sirouspour. 2014. "GPU-based acceleration of computations in nonlinear finite element deformation analysis." *Int. J. Numer. Methods Biomed. Eng.* 30 (3): 365–381. https://doi.org/10.1002/cnm.2607.

Markall, G., A. Slemmer, D. Ham, P. Kelly, C. Cantwell, and S. Sherwin. 2013. "Finite element assembly strategies on multi-core and many-core architectures." *Int. J. Numer. Methods Fluids* 71 (1): 80–97. https://doi.org/10.1002/fld.3648.

Martínez-Frutos, J., P. J. Martínez-Castejón, and D. Herrero-Pérez. 2015. "Fine-grained GPU implementation of assembly-free iterative solver for finite element problems." *Comput. Struct.* 157 (Sep): 9–18. https://doi.org/10.1016/j.compstruc.2015.05.010.

McCallen, D., A. Petersson, A. Rodgers, A. Pitarka, M. Miah, F. Petrone, B. Sjogreen, N. Abrahamson, and H. Tang. 2021a. "EQSIM—A multidisciplinary framework for fault-to-structure earthquake simulations on exascale computers part I: Computational models and workflow." *Earthquake Spectra* 37 (2): 707–735. https://doi.org/10.1177/8755293020970982.

McCallen, D., F. Petrone, M. Miah, A. Pitarka, A. Rodgers, and N. Abrahamson. 2021b. "EQSIM—A multidisciplinary framework for

fault-to-structure earthquake simulations on exascale computers, Part II: Regional simulations of building response." *Earthquake Spectra* 37 (2): 736–761. https://doi.org/10.1177/8755293020970980.

McCallen, D., H. Tang, S. Wu, E. Eckert, J. Huang, and N. A. Petersson. 2022. "Coupling of regional geophysics and local soil-structure models in the EQSIM fault-to-structure earthquake simulation framework." *Int. J. High Perform. Comput. Appl.* 36 (1): 78–92. https://doi.org/10.1177/10943420211019118.

McKenna, F. 1997. *Object-oriented finite element programming: Frameworks for analysis, algorithms and parallel computing*. Berkeley, CA: Univ. of California.

McKenna, F., M. H. Scott, and G. L. Fenves. 2010. "Nonlinear finite-element analysis software architecture using object composition." *J. Comput. Civ. Eng.* 24 (1): 95–107. https://doi.org/10.1061/(ASCE)CP.1943-5487.0000002.

Mihaila, B., M. Knezevic, and A. Cardenas. 2014. "Three orders of magnitude improved efficiency with high-performance spectral crystal plasticity on GPU platforms." *Int. J. Numer. Methods Eng.* 97 (11): 785–798. https://doi.org/10.1002/nme.4592.

Mills, R. T., et al. 2021. "Toward performance-portable PETSc for GPU-based exascale systems." *Parallel Comput.* 108 (2021): 102831. https://doi.org/10.48550/arXiv.2011.00715.

Motley, M. R., H. K. Wong, X. Qin, A. O. Winter, and M. O. Eberhard. 2016. "Tsunami-induced forces on skewed bridges." *J. Waterway, Port, Coastal, Ocean Eng.* 142 (3): 04015025. https://doi.org/10.1061/(ASCE)WW.1943-5460.0000328.

Naumov, M., et al. 2015. "AmgX: A library for GPU accelerated algebraic multigrid and preconditioned iterative methods." *SIAM J. Sci. Comput.* 37 (5): S602–S626. https://doi.org/10.1137/140980260.

Neic, A., M. Liebmann, and G. Haase. 2012. "Algebraic multigrid solver on clusters of CPUs and GPUs." In *Proc., Int. Workshop on Applied Parallel and Scienti c Computing*, 389–398. Berlin: Springer.

Newmark, N. 1959. "A method of computation for structural dynamics." *J. Eng. Mech. Div.* 85 (3): 67–94. https://doi.org/10.1061/JMCEA3.0000098.

NVIDIA. 2007. *Compute unified device architecture programming guide*. Santa Clara, CA: NVIDIA.

NVIDIA. 2012. *PhysX*. Santa Clara, CA: NVIDIA.

NVIDIA. 2013. *NVIDIA CUDA C programming guide*. Santa Clara, CA: NVIDIA.

NVIDIA. 2014. *CuSP*. Santa Clara, CA: NVIDIA.

NVIDIA. 2015. *GPU-accelerated applications*. Santa Clara, CA: NVIDIA.

NVIDIA. 2019. *CUDA C best practices guide (v5.0)*. Santa Clara, CA: NVIDIA.

Nyland, L., M. Harris, and J. Prins. 2007. "Fast N-body simulation with CUDA." In *GPU Gems 3*, 677–695. Boston: Addison-Wesley.

O'Reilly, O., T.-Y. Yeh, K. B. Olsen, Z. Hu, A. Breuer, D. Roten, and C. A. Goulet. 2022. "A high-order finite-difference method on staggered curvilinear grids for seismic wave propagation applications with topography." *Bull. Seismol. Soc. Am.* 112 (1): 3–22. https://doi.org/10.1785/0120210096.

O'Rourke, T. D. 2010. "Geohazards and large, geographically distributed systems." *Géotechnique* 60 (7): 505–543. https://doi.org/10.1680/geot.2010.60.7.505.

Owens, J. D., D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. 2005. "A survey of general-purpose computation on graphics hardware." *Comput. Graphics Forum* 26 (1): 80–113. https://doi.org/10.1111/j.1467-8659.2007.01012.x.

Papadrakakis, M., G. Stavroulakis, and A. Karatarakis. 2011. "A new era in scientific computing: Domain decomposition methods in hybrid CPU-GPU architectures." *Comput. Methods Appl. Mech. Eng.* 200 (13–16): 1490–1508. https://doi.org/10.1016/j.cma.2011.01.013.

Peterson, B., A. Humphrey, J. Holmen, T. Harman, M. Berzins, D. Sunderland, and H. C. Edwards. 2018. "Demonstrating GPU code portability and scalability for radiative heat transfer computations." *J. Comput. Sci.* 27 (Jul): 303–319. https://doi.org/10.1016/j.jocs.2018.06.005.

Posey, S., and F. Courteille. 2012. "GPU progress in sparse matrix solvers for applications in computational mechanics." In Vol. ESCO12 of *Proc., European Seminar on Computing*. Reston, VA: American Institute of Aeronautics and Astronautics.

Roa, M., K. Logarathan, and N. V. Raman. 1994. "Multi-frontal based approach for concurrent finite element analysis." *Comput. Struct.* 52 (4): 841–846. https://doi.org/10.1016/0045-7949(94)90364-6.

Roten, D., Y. Cui, K. Olsen, S. Day, K. Withers, W. Savran, P. Wang, and D. Mu. 2016. "High-frequency nonlinear earthquake simulations on petascale heterogeneous supercomputers." In *Proc., SC'16 Proc. Supercomputing Conf.* New York: IEEE.

Rumpf, M., and R. Strzodka. 2005. "Numerical solution of partial differential equations on parallel computers." In Vol. 51 of *Lecture notes in computational science and engineering*, edited by A. M. Bruaset and A. Tveito, 89–134. Berlin: Springer.

Santiago, E. D., and K. H. Law. 1996. "An implementation of finite element method on distributed workstations." In *Proc., Analysis and Computation: Proc. of the Twelfth Conf. Held in Conjunction with Structures Congress XIV,* edited by F. Y. Cheng, 188–199. Reston, VA: ASCE.

Schenk, O., M. Christen, and H. Burkhart. 2008. "Algorithmic performance studies on graphics processing units." *J. Parallel Distrib. Comput.* 68 (10): 1360–1369. https://doi.org/10.1016/j.jpdc.2008.05.008.

Sharma, G., A. Agarwala, and B. Bhattacharya. 2013. "A fast parallel Gauss Jordan algorithm for matrix inversion using CUDA." *Comput. Struct.* 128 (Nov): 31–37. https://doi.org/10.1016/j.compstruc.2013.06.015.

Siegel, A., E. Draeger, J. Deslippe, A. Dubey, T. Evans, T. Germann, and W. Hart. 2020. *Early application results on pre-exascale architecture with analysis of performance challenges and projections*. Washington, DC: Exascale Computing Project.

Snell, A., and L. Segervall. 2017. *HPC application support for GPU computing*. Sunnyvale, CA: Intersect 360 Research: Accurate Market Intelligence for High Performance Computing.

Stone, C. P., and R. L. Davis. 2013. "Techniques for solving stiff chemical kinetics on graphical processing units." *J. Propul. Power* 29 (4): 764–773. https://doi.org/10.2514/1.B34874.

Sunarso, A., T. Tsuji, and S. Chono. 2010. "GPU-accelerated molecular dynamics simulation for study of liquid crystalline flows." *J. Comput. Phys.* 229 (15): 5486–5497. https://doi.org/10.1016/j.jcp.2010.03.047.

Sutter, H. 2005. "The free lunch is over: A fundamental turn toward concurrency in software." *Dr. Dobb's J.* 30 (3): 202–210.

Sylwestrzak, M., D. Szlag, P. J. Marchand, A. S. Kumar, and T. Lasser. 2017. "Massively parallel data processing for quantitative total flow imaging with optical coherence microscopy and tomography." *Comput. Phys. Commun.* 217 (Aug): 128–137. https://doi.org/10.1016/j.cpc.2017.03.008.

Synn, S. Y., and R. E. Fulton. 1995. "Practical strategy for concurrent substructure analysis." *Comput. Struct.* 54 (5): 939–944. https://doi.org/10.1016/0045-7949(94)00385-G.

Tavakkol, S., and P. Lynett. 2017. "Celeris: A GPU-accelerated open source software with a Boussinesq-type wave solver for real-time interactive simulation and visualization." *Comput. Phys. Commun.* 217 (Aug): 117–127. https://doi.org/10.1016/j.cpc.2017.03.002.

Taylor, Z., M. Cheng, and S. Ourselin. 2008. "High-speed nonlinear finite element analysis for surgical simulation using graphics processing units." *IEEE Trans. Med. Imaging* 27 (5): 650–663. https://doi.org/10.1109/TMI.2007.913112.

Tian, Y., L. Xie, Z. Xu, and X. Lu. 2015. "GPU-powered high-performance computing for the analysis of large-scale structures based on OpenSees." In *Proc., ASCE Computing in Civil Engineering*, 411–418. Reston, VA: ASCE.

Ting, E. C., C. Shih, and Y.-K. Wang. 2004. "Fundamentals of a vector form intrinsic finite element: Part I. Basic procedure and a plane frame element." *J. Mech.* 20 (2): 113–122. https://doi.org/10.1017/S1727719100003336.

Tomov, S., J. Dongarra, and M. Baboulin. 2010. "Towards dense linear algebra for hybrid GPU accelerated manycore systems." *Parallel Comput.* 36 (5–6): 232–240. https://doi.org/10.1016/j.parco.2009.12.005.

Topping, B. H., and A. I. Khan. 1996. *Parallel finite element computations*. Edinburgh, UK: Saxe-Coburg Publications.

Verschoor, M., and A. C. Jalba. 2012. "Analysis and performance estimation of the conjugate gradient method on multiple GPUs." *Parallel Comput.* 38 (10–11): 552–575. https://doi.org/10.1016/j.parco.2012.07.002.

Wagner, M., K. Rupp, and J. Weinbub. 2012. "A comparison of algebraic multigrid preconditioners using graphics processing units and multicore central processing units." In Vol. 2 of *Proc., 2012 Symp. on High Performance Computing, HPC '12*, 1–8. San Diego: Society for Computer Simulation International.

Wang, M., H. Klie, M. Parashar, and H. Sudan. 2009. "Solving sparse linear systems on NVIDIA Tesla GPUs." In *Proc., Computational Science (ICCS 2009)*, 864–873. Cham, Switzerland: Springer Nature.

Yamaguchi, T., K. Fujita, T. Ichimura, A. Naruse, M. Lalith, and M. Hori. 2020. "GPU implementation of a sophisticated implicit low-order finite element solver with FP21-32-64 computation using OpenACC." In Vol. 12017 of *Lecture notes in computer science (including subseries lecture notes in artificial intelligence and lecture notes in bioinformatics)*, 3–24. Cham, Switzerland: Springer.

Yang, Y.-S., C.-M. Yang, and T.-J. Hsieh. 2014. "GPU parallelization of an object-oriented nonlinear dynamic structural analysis platform." *Simul. Modell. Pract. Theory* 40 (Jan): 112–121. https://doi.org/10.1016/j.simpat.2013.09.004.

Zhang, W., and E. M. Lui. 1991. "A parallel frontal solver on the Alliant FX/80." *Comput. Struct.* 38 (2): 203–215. https://doi.org/10.1016/0045-7949(91)90097-6.

Zhou, H., G. Mo, F. Wu, J. Zhao, M. Rui, and K. Cen. 2012. "GPU implementation of lattice Boltzmann method for flows with curved boundaries." *Comput. Methods Appl. Mech. Eng.* 225 (Jun): 65–73. https://doi.org/10.1016/j.cma.2012.03.011.

Zhou, J., Y. Cui, E. Poyraz, D. J. Choi, and C. C. Guest. 2013. "Multi-GPU implementation of a 3D finite difference time domain earthquake code on heterogeneous supercomputers." *Procedia Comput. Sci.* 18 (Jan): 1255–1264. https://doi.org/10.1016/j.procs.2013.05.292.

Zhu, M., and M. H. Scott. 2014. "Modeling fluid–structure interaction by the particle finite element method in OpenSees." *Comput. Struct.* 132 (Feb): 12–21. https://doi.org/10.1016/j.compstruc.2013.11.002.

Zienkiewicz, O. C., and R. L. Taylor. 1989. *The finite element method*. London: McGraw-Hill.