CHAPTER

16

Application of Assembly of Finite Element Methods on Graphics Processors for Real-Time Elastodynamics

Cris Cecka, Adrian Lew, and Eric Darve

In this chapter, we discuss multiple strategies to perform general computations on unstructured grids, with specific application to the assembly of matrices in finite element methods (FEMs). We review and apply two methods, discussed in depth in [1], for assembly of FEMs to produce and accelerate a FEM model for a nonlinear hyperelastic solid where the assembly, solution, update, and visualization stages are performed solely on the GPU, benefiting from speed-ups in each stage and avoiding costly GPU-CPU transfers of data. For each method, we discuss the NVIDIA GPU hardware's limiting resources, optimizations, key data structures, and dependence of the performance with respect to problem size, element size, and GPU hardware generation. This chapter will inform potential users of the benefits of GPU technology, provide guidelines to help them implement their own FEM solutions, give potential speed-ups that can be expected, and provide source code for reference.

16.1 INTRODUCTION, PROBLEM STATEMENT, AND CONTEXT

In the domain of partial differential equations (PDEs), finite difference methods naturally fit into the GPU computing environment. Finite difference approaches have regular, vectorizable data access patterns, making them a natural candidate for execution on GPUs. When solving PDEs in complex domains or when adaptive mesh refinement strategies are needed, unstructured grids are often more convenient and the method of finite elements is appealing. In this section, we introduce elementary notions on finite elements and the important primitive operations and data flow. For more detail, we recommend any FEM textbook, such as [2].

Unstructured meshes are common in many engineering and graphics applications to create versatile discretizations of PDEs. To illustrate this, consider the problem of finding a function $u\colon \Omega \to \mathbb{R}$ that satisfies

$$\mathcal{L}u = f \quad \text{in} \quad \Omega \tag{1}$$

and subject to some boundary conditions. Here, Ω is a domain, $\mathcal{L}$ is a general linear differential operator, and f is a scalar-valued function over Ω.

GPU Computing Gems

A standard finite element method begins by constructing a finite dimensional space $\mathcal{V}_h$ of functions over Ω. The numerical approximation of the solution is then written as

$$u_h(\boldsymbol{x}) = \sum_j u_j \varphi_j(\boldsymbol{x}),$$

where $\{u_j\}$ are the components of u_h in the basis $\{\varphi_j\}$ of $\mathcal{V}_h$. Then Eq. (1) is multiplied by φ_i and integrated over Ω to obtain a weak formulation:

$$a(u_h, \varphi_i) := \sum_j u_j \int_\Omega \varphi_i \, \mathcal{L}\varphi_j \, d\Omega = \int_\Omega \varphi_i f \, d\Omega, \qquad \forall \varphi_i. \tag{2}$$

These equations are then further transformed (using integration by parts, the boundary conditions of the problem, etc.) to yield a linear system of equations:

$$\boldsymbol{A}\boldsymbol{u} = \boldsymbol{F},$$

where $\boldsymbol{A}$ is the so-called stiffness matrix and $\boldsymbol{F}$ is the forcing vector. The numerical approximation, u_h, follows by solving this system for the vector of components $\boldsymbol{u} = [u_i]$.

In the finite element method, the basis functions $\{\varphi_i\}$ are constructed using a partition of the domain into a set $\mathcal{E}$ of disjoint domains, $\Omega^e \subset \Omega$, $e \in \mathcal{E}$, termed *elements* (see Figure 16.1). Then, the bilinear form defined in Eq. (2) can be split as

$$A_{ij} = a(\varphi_i, \varphi_j) = \sum_{e \in \mathcal{E}} a^e(\varphi_i, \varphi_j),$$

where a^e is the bilinear form that results from restricting the integral in Eq. (2) to Ω^e.

Typically, finite element formulations introduce a set, $\mathcal{N}$, of nodes $\boldsymbol{x}_i \in \Omega$, and basis functions are often chosen such that $\varphi_j(\boldsymbol{x}_i) = \delta_{ij}$, so that u_h is uniquely determined by its values at the nodes. In the simplest case, the set $\mathcal{N}$ is the set of vertices of the polyhedral elements. Furthermore, in most cases, the finite element basis functions $\{\varphi_i\}$ are nonzero only over the elements containing $\boldsymbol{x}_i$. Then, the elemental contribution

$$A^e_{ij} = a^e(\varphi_i, \varphi_j)$$

to A_{ij} from element $e \in \mathcal{E}$ is different from zero if and only if the points $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ are both in or on the boundary of e. Consequently, during assembly A^e_{ij} needs to be computed only for a few values of i and j, independent of the size of the mesh. These elemental contributions are then accumulated into matrix $\boldsymbol{A}$.

Usually, the computation of the numerical solution is performed in two steps: the assembly of matrix $\boldsymbol{A}$ and vector $\boldsymbol{F}$, and the solution of the linear system $\boldsymbol{A}\boldsymbol{u} = \boldsymbol{F}$. Matrix-free iterative methods, which do not assemble and store the system explicitly, are advocated in [3]. Many applications require the assembly and solution to be performed many times, such as when $\mathcal{L}$ is nonlinear and/or time-dependent.

Although the computational cost of the assembly procedure is smaller than the computational cost of solving the resulting system of equations, we show in this chapter that with emerging research in adapting the conjugate gradient method [4–6] and sparse matrix-vector multiplication (SpMV) [7–10]

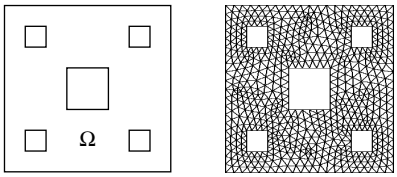


FIGURE 16.1

Illustration of a domain and associated finite element mesh. Each vertex is a node and each triangle is an element of the finite element mesh.

to the GPU, the assembly stage becomes a bottleneck. Devising and implementing algorithms for FEM assembly on the GPU not only prevents costly CPU-GPU transfers of data, but can also provide a significant speed-up to the application.

16.2 CORE METHOD

16.2.1 Finite Element Assembly

The assembly procedure partitions the integrals over Ω in Eq. (2) as a sum of integrals over the elements, so that each entry A_{ij} or F_i is computed as a sum of elemental contributions,

$$A_{ij} = a(\varphi_i, \varphi_j) = \sum_{e \in \mathcal{E}} a^e(\varphi_i, \varphi_j) = \sum_{e \in \mathcal{E}} A^e_{ij},$$

A typical finite element assembly program relies on given *element subroutines* to compute element matrices $\boldsymbol{A}^{\mathrm{e}}$ and element forcing vectors $\boldsymbol{F}^{\mathrm{e}}$. These element subroutines change with the PDE, the element type, and the basis functions, and are functions of the nodal coordinates, any nodal fields, forces, or boundary conditions, and other element parameters.

Three types of data structures are then important for computations over unstructured meshes:

1. The nodal data matrix $C(n)$, which yields the field values of the n^{th} node, with n, $0 \leq n < |\mathcal{N}|$, referred to as the *global node number*. The first fields are often the coordinates of the node, followed by nodal values of other fields, such as a force. In the case of nonlinear problems, the system of equations depends on u. Hence, nodal data will also include the values of the required degrees of freedom, such as temperature or displacement.
2. The supplemental data matrix $S(e)$, which yields supplemental data for the e^{th} element. For example, inhomogeneous problems may require spatially varying element subroutines that are parameterized by some value, i.e., permeability, conductivity, material constants, etc.
3. The connectivity matrix $E(e,a)$, which yields the global node number of the a^{th} node of the e^{th} element. Here, e, $0 \leq e < |\mathcal{E}|$, is referred to as the *global element number* and a, $0 \leq a < e_n$, where e_n denotes the number of nodes per elements, is referred to as the *local node number*.

The input arguments for element e are the nodal data contained in $C(n)$, for each global node number n in the element, and the supplemental data $S(e)$. The nodal data is generally retrieved from

Algorithm 1: The direct stiffness method of finite element assembly

```
1: Initialize A and F to zero;
2: for all elements e ∈ E do
3:     (A^e, F^e) ← elem(e) ;                          /* element subroutine */
4:     for all local degrees of freedom d_1 of e do
5:         F(L(e,d_1)) += F^e(d_1);
6:         for all local degrees of freedom d_2 of e do
7:             A(L(e,d_1),L(e,d_2)) += A^e(d_1,d_2);
```

memory and arranged following a local node numbering scheme for the element. Similarly, after the element data $\boldsymbol{A}^{\mathrm{e}}$ and $\boldsymbol{F}^{\mathrm{e}}$ are computed, these data are accumulated in $\boldsymbol{A}$ and $\boldsymbol{F}$ following a map from local to global degrees of freedom.

This mapping information is stored in the *location matrix L*. For the d^{th} degree of freedom of the e^{th} element, $L(e,d)$ is the corresponding global degree of freedom number. This mapping allows us to write and store $\boldsymbol{A}^{\mathrm{e}}$ and $\boldsymbol{F}^{\mathrm{e}}$ densely so that

$$\boldsymbol{A}(i,j) = \sum_{\substack{e,d_1,d_2 \\ L(e,d_1)=i \\ L(e,d_2)=j}} \boldsymbol{A}^{\mathrm{e}}(d_1,d_2) \qquad \boldsymbol{F}(i) = \sum_{\substack{e,d \\ L(e,d)=i}} \boldsymbol{F}^{\mathrm{e}}(d), \tag{3}$$

where we have adopted the notation $A_{ij} = \boldsymbol{A}(i,j)$ and $F_i = \boldsymbol{F}(i)$. An implementation is given in Algorithm 1. This is known as the direct stiffness method and is the most common implementation of finite element assembly.

16.3 ALGORITHMS, IMPLEMENTATIONS, AND EVALUATIONS

Previous studies on finite element methods (FEMs) on GPUs have largely focused on the solution of the sparse linear system of equations resulting from a FEM discretization [4, 6, 11], mainly because the solution stage is often the most computationally intensive step. Some assembly strategies for the GPU have been mentioned as well. However, specific applications have often allowed special approaches for FEM assembly. The methods, described in [11] for geometric flow on an unstructured mesh and in [12] for FEM cloth simulation, derive relatively simple expressions for each nonzero entry in the system of equations. This relative simplicity and the inherent parallelism of computing each nonzero entry independently makes these approaches well suited to GPUs.

In this section, we review, update, and apply methods for finite element assembly from [1] that are general enough to be used for a wide range of finite element models. Consequently, we attempt to make few, unrestrictive assumptions about the properties of the problem, the element subroutine, and the sparse matrix format. The following assumptions are nonetheless made:

- The element subroutine is provided as a black box intended to be executed by a single thread on the GPU. This may be the case for a wide range of low-order finite element methods that are

ideal for acceleration. Problem-dependent optimizations like those found in [11, 13, 14] and high-order optimizations such as parallelizing the element subroutine [15] can also be investigated for additional performance improvement.

- The sparse matrix format provides a one-to-one mapping $K : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ which takes a row-column pair, (i,j), corresponding to a nonzero of the matrix and returns the index into a value array where this nonzero entry is stored.
- The connectivity, $E(e,a)$, and supplemental data, $S(e)$, are constant over the course of the computation. This allows significant precomputation to be performed, which greatly accelerates the methods. This prevents the easy application of these methods to dynamically changing or adaptive meshes.

The first method, in Section 16.3.1, consists of decomposing the calculation into two phases: all the element data are computed and written to global memory. It is then assembled into the matrix $\boldsymbol{A}$ in any number of ways. The advantages of this method include its relative ease of implementation and its potential for further improvement since it is primarily limited by the second stage: the reduction of the element data into the system of equations.

The second method, in Section 16.3.2, is more complex and uses the shared memory space to stage the element data and reduce the number of transactions with global memory. The nodes and elements are partitioned into subdomains wherein some set of nonzeros of the system of equations can be safely computed. A thread block computes all the element data required for a subdomain and then accumulates and assembles all possible nonzero entries in the matrix $\boldsymbol{A}$ and vector $\boldsymbol{F}$. This can reduce the number of passes through global memory while avoiding excessive recomputation of the element data. However, this method is restricted by the size of the shared memory space, the size of the element data $\boldsymbol{A}^{\mathrm{e}}$ and $\boldsymbol{F}^{\mathrm{e}}$, and the connectivity of each node.

Both of these methods are slightly modified from those found in [1], where more detail on each method may be found as well as additional methods not considered in this chapter. In this chapter, the methods are updated to account for the need to include supplemental data, $S(e)$ — that are associated with an element rather than a node. The supplemental data appear in the application of these methods to a nonlinear elastic model at the end of this chapter. Additionally, we discuss the use of these methods on more modern hardware than was available for [1], and how this affects their performance.

16.3.1 Assembly by Nonzero Entries Using Global Memory

This approach, deemed GlobalNZ, assigns one thread to compute the element data for one element at a time and, to avoid race conditions, writes the element data in coalesced memory transactions to global memory for later reduction into the system of equations. Since the reduction stage then operates on element data stored in global memory and there are no global synchronization primitives, the computation and assembly of the element data must be performed using separate kernels.

A significant optimization is made by exploiting the fact that elements can be grouped to share many nodes. The total number of transactions with global memory can be reduced by prefetching all the nodal data a thread block will require and sharing it between the set of elements to be computed.

Thus, we precompute the set of nodes $\mathcal{N}_k$ that thread block k will require for all the elements $\mathcal{E}_k$ that it is responsible for computing. The nodal data corresponding to the nodes of $\mathcal{N}_k$ are prefetched into shared memory. Each thread is then assigned to compute the element data for an element using the nodal data in shared memory. To do this, each thread reads from a precomputed array telling it which

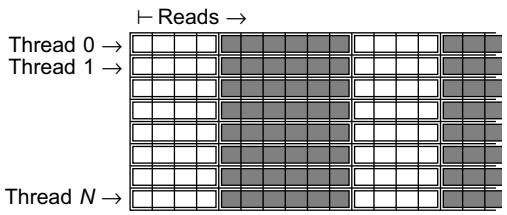


FIGURE 16.2

The column-major block element matrix E_k in the case $e_n = 4$. Each white entry stores a block node number $E_k(e,a)$, which can be used to find the nodal data in shared memory. Each group of 4 entries defines an element to be computed. Followed, in gray, are any constant supplemental data to be given to the element. A thread block will read down a column of the array in coalesced memory transactions.

nodes to retrieve from shared memory. That is, for each thread block k, we precompute block element matrices, E_k, defined by

$$E_k(e,a) = \sigma_k(E(e,a)) \qquad \forall\, e \in \mathcal{E}_k,\ a = 1,\ldots,e_n$$

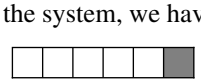
where $\sigma_k : \mathcal{N}_k \to \{1,\ldots,|\mathcal{N}_k|\}$ is a mapping of global node number $E(e,a)$ to block node number $E_k(e,a)$ within block k, which can be used to find the nodal data in shared memory. Although not considered in [1], constant supplemental data should also be appended to this list. This results in the data structure shown in Figure 16.2. A thread will read $e_n + |S(e)|$ values, where e_n is the number of nodes per element, in coalesced memory transactions. It then passes to the element subroutine the supplemental data and indices into shared memory pointing to the required nodal data. The element subroutine computes the element data and stores it into global memory using coalesced memory writes.

It is mentioned in [1] that the assembly of the element data from global memory can be expressed as the sparse matrix vector multiplication

$$[\boldsymbol{A}\,;\boldsymbol{F}] = \boldsymbol{S}\boldsymbol{G}$$

where $\boldsymbol{A}$ and $\boldsymbol{F}$ define the system of equations, $\boldsymbol{G}$ is the element data stored in global memory, and $S_{ij} \in \{0,1\}$ is a matrix appropriately constructed to perform the summation in Eq. (3). Thus, this stage of the global assembly method should take advantage of the emerging research in sparse matrix-vector multiplication (SpMV) methods on GPUs. However, it should be noted that in practical tests, we find that the reduction array method presented in [1] outperforms many SpMV routines, including CSR, HYB, and ELLPACK-R, to preform this reduction. This may be simply due to the superfluous lookup of the "1"s in the SpMVs, and this is under ongoing research.

Our approach to the reduction step involves determining the indices into the element data array previously computed and stored in global memory that contribute to each nonzero entry (NZ) of the system of equations. Each one of these lists is appended with the index into the system of equations of the NZ in question. Thus, for each NZ of the system, we have an NZ reduction list of the form



where the light entries represent indices into the element data array (source indices) and the final dark entry is the index of the NZ in the system of equations (target index), which can be distinguished by the sign of the integer. Each NZ of the system of equations has an associated reduction list of this form. However, the lists may have significantly differing lengths. Assigning one thread to one list could lead to unbalanced workloads. Furthermore, we need to coalesce the access to these lists. To efficiently perform these reduction operations in parallel, we pack these lists into a *reduction array*. First, we decide the number of NZs to compute per block. For each block, we then pack the NZ reduction lists into an array that will be read fully coalesced in the kernel. Simple packing algorithms such as Largest-Processing-Time (LPT) [16] appear to suffice and result in only small amounts of wasted space provided there are enough reduction lists per block. The result is a column-major matrix with *blockSize* rows and data profile shown in Figure 16.3 and the parallel algorithm using this structure is given in Algorithm 2.

For efficiency and simplicity, we assemble contiguous entries of the matrix value array in shared memory. That is, we map contiguous entries of a matrix to the shared memory space, and perform coalesced writes into matrix's value array when all NZs have been computed. Thus, the only memory reads

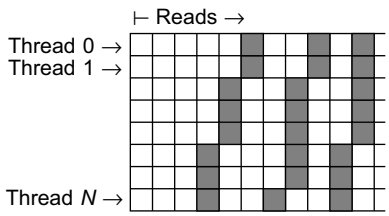


FIGURE 16.3

The column-major reduction matrix with *blockSize* rows. The light entries represent source indices and the dark entries represent target indices. A thread block will read down a column of the array in coalesced memory transactions.

Algorithm 2: Reduction operation from reduction array. In this case, the source array S is the element data array in global memory and the target array T is the system of equations stored in global memory.

```
1:  k ← blockID;
2:  tid ← blockThreadID;
3:  t ← 0;
4:  while tid < end_k do
5:      i ← reductionArray_k[tid];
6:      if i > 0 then
7:          t ← t + S[i − 1];
8:      else if i < 0 then
9:          T[−i − 1] ← t;
10:         t ← 0;
11:     tid ← tid + blockSize;
```

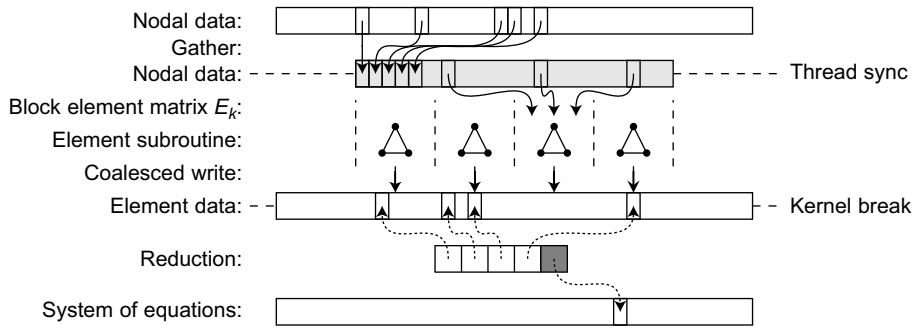


FIGURE 16.4

The global assembly by NZ on the GPU. Global memory is depicted in white and shared memory is depicted in light gray. Solid black arrows represent memory reads and writes and dotted arrows represent references to memory. The kernel break denotes the end of the element computation kernel and the beginning of the assembly kernel.

that are not coalesced are the retrievals of the element data. This approach proves to be quite efficient, but also causes a slight dependency on the storage structure of the sparse matrix. If there is significant padding in the sparse matrix's value array (presumably for efficient matrix-vector multiplication), then this approach will be uselessly writing these values as well as the NZs. However, the matrix format is presumably designed to be accessed in coalesced reads in order to optimize the SpMV kernel. Assembling the NZs in this same access pattern could be a significant optimization to this stage — although this is difficult to recognize and perform in the general case.

The entire assembly algorithm by NZ using global memory is diagrammed in Figure 16.4.

16.3.2 Assembly by Nonzero Entries Using Shared Memory

The second approach, deemed SharedNZ, uses shared memory to stage the element data and perform the assembly/reduction step on-chip. First, a thread block is assigned responsibility for assembling a set of nonzero entries (NZs) of the system of equations. For the NZs to be assembled from element data in shared memory, we must guarantee that all of the element data that contribute to the set of NZs are available in shared memory.

Thus, we first partition the nodes and assign a thread block to be responsible for assembling all NZs associated with a node in that partition. Although [1] recommends METIS [17] to perform the partition, METIS failed in [1] to define partitions small enough to allow the 5th order triangular element test case to proceed. We find METIS to be insufficient for the partitioning that SharedNZ requires and, instead, wrote a simple greedy algorithm to determine nodal partitions. This greedy algorithm is initialized with each partition containing one node. At each iteration, a node finds an adjacent partition which, if added to its partition, would result in the smallest number of elements that would be required to assemble all nodes of the unioned partition. If the shared memory constraint is satisfied, the two partitions are unioned and we move on to the next node until no more partitions can be unioned. We find this acceptable and note that it actually outperforms METIS's partitions in most cases.

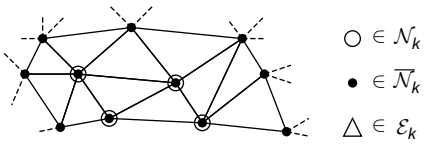


FIGURE 16.5

Example of sets $\overline{\mathcal{N}}_k$, $\mathcal{N}_k$, and $\mathcal{E}_k$. As $|\mathcal{E}_k|$ increases, the ratio $|\overline{\mathcal{N}}_k| / |\mathcal{N}_k|$ decreases, which improves the efficiency of the algorithm.

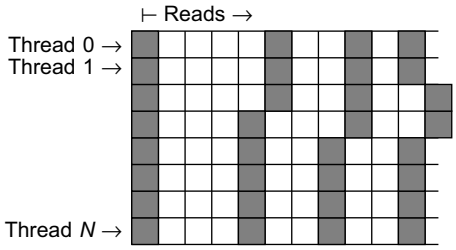
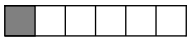


FIGURE 16.6

The column-major scatter matrix with *blockSize* rows. The dark entries represent source indices and the light entries represent target indices. A thread block will read a column of the array in coalesced memory transactions.

For thread block k, let $\mathcal{N}_k$ be the set of nodes it is responsible for assembling. We denote $\mathcal{E}_k$ the set of elements required to compute the associated NZs. In order to compute elemental data for $\mathcal{E}_k$, a thread block must retrieve the data for any node adjacent to elements of $\mathcal{E}_k$. We call this set of nodes $\overline{\mathcal{N}}_k$. Figure 16.5 shows an example of the sets $\mathcal{N}_k$, $\overline{\mathcal{N}}_k$, and $\mathcal{E}_k$.

The nodes of $\overline{\mathcal{N}}_k$ are to be fetched from global memory once and stored in shared memory as input for the element subroutines. For each node in $\overline{\mathcal{N}}_k$, we define the list of elements in $\mathcal{E}_k$ that require the data from that node. For each node we create a list of the form



where the dark entry represents the node number (source index) to be retrieved from global memory and the light entries represent the block element number that requires the data from that node (target index). Again, the lists associated to different nodes in $\overline{\mathcal{N}}_k$ may have significantly different lengths, but we can pack these lists into a *scatter array* using a packing algorithm such as LPT [16]. The result is a column-major matrix with *blockSize* rows and data profile shown in Figure 16.6. The parallel algorithm using this structure is similar to Algorithm 2.

In addition to the nodal data, the element subroutines also require the element-wise supplemental data. In contrast to the approach taken in Section 16.3.1, we have simply adopted computing a separate array in global memory of the supplemental data that are ordered such that they can be read by the elements with coalesced reads.

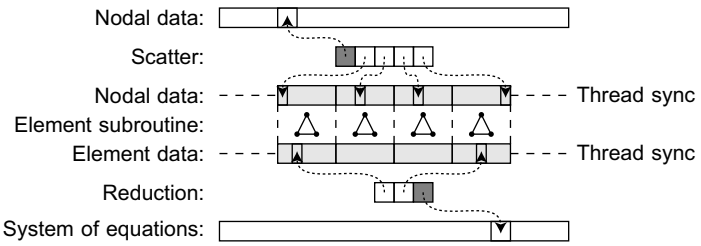


FIGURE 16.7

The shared assembly by NZ on the GPU. Global memory is depicted in white and shared memory is depicted in light gray. Dotted arrows represent references into memory.

After the element subroutines have calculated the element data and stored them into shared memory (overwriting the nodal data), we need to assemble them into the system of equations. By construction, all of the element data needed by a set of NZs of the system of equations are now in the shared memory space and we can use a similar approach to the scatter operation in the previous section to perform the reduction operation.

The entire procedure is diagrammed in Figure 16.7. Note that the shared assembly by NZ algorithm is heavily constrained by the size of the shared memory space.

16.4 EVALUATION AND VALIDATION OF RESULTS, TOTAL BENEFITS, LIMITATIONS

16.4.1 Benchmark Problem — Nonlinear Elastodynamics

We wish to apply the above methods for finite element assembly to a realistic problem which depends critically on the efficiency of the finite element assembly stage. For this purpose, we chose to simulate the dynamics of a nonlinear elastic model in three dimensions; see, e.g., [18]. The essential details of this model are introduced below.

We consider a body occupying a reference configuration $\Omega \subset \mathbb{R}^3$. The motion of this body is described through a smooth map $\phi\colon \Omega \times \mathbb{R} \to \mathbb{R}^3$, so that $\phi(\mathbf{X},t)$ describes the position at time t of the material particle at $\mathbf{X} \in \Omega$. This motion may be a consequence of the elasticity of the body, of forces or constraints acting on $\partial\Omega$, or of a system of body forces $\boldsymbol{b}\colon \Omega \times \mathbb{R} \to \mathbb{R}^3$. The elasticity of the body is described by the strain energy density per unit volume of a neo-Hookean material extended to the compressible range, namely,

$$W(\boldsymbol{\mathcal{F}};\boldsymbol{X}) := \frac{\mu}{2}(\boldsymbol{\mathcal{F}}^T\boldsymbol{\mathcal{F}} - \boldsymbol{I}) : \boldsymbol{I} + \frac{\lambda}{2}\ln^2(J) - \mu\ln(J),$$

where $\boldsymbol{\mathcal{F}} = \nabla\phi(\boldsymbol{X},t)$ is the spatial gradient of ϕ, or deformation gradient, μ and λ are elastic constants, and J is the determinant of $\boldsymbol{\mathcal{F}}$. Then the first Piola-Kirchhoff stress tensor is given by

$$\boldsymbol{P} := \mu\boldsymbol{\mathcal{F}} + (\lambda\ln(J) - \mu)\boldsymbol{\mathcal{F}}^{-T}.$$

Variational time-integrators for these systems are discussed, for example, in [19]. The traditional midpoint rule is an example of such an integrator, and takes the form

$$\boldsymbol{p}^k = \boldsymbol{m}\frac{\boldsymbol{\phi}^{k+1}-\boldsymbol{\phi}^k}{\Delta t} + \frac{\Delta t}{2}\boldsymbol{f}\left(\frac{\boldsymbol{\phi}^{k+1}+\boldsymbol{\phi}^k}{2}\right) =: \boldsymbol{h}(\boldsymbol{\phi}^{k+1}) \tag{4}$$

$$\boldsymbol{p}^{k+1} = \boldsymbol{m}\frac{\boldsymbol{\phi}^{k+1}-\boldsymbol{\phi}^k}{\Delta t} - \frac{\Delta t}{2}\boldsymbol{f}\left(\frac{\boldsymbol{\phi}^{k+1}+\boldsymbol{\phi}^k}{2}\right) \tag{5}$$

where $\boldsymbol{p}^k$ and $\boldsymbol{\phi}^k$ are the vector of nodal momentum and positions at time t^k, and $\boldsymbol{m}$ and $\boldsymbol{f}$ are the mass matrix and the force vector given by

$$(\boldsymbol{m})_{a_i b_j} = \int_\Omega \rho\, \varphi_a\, \varphi_b\, d\Omega \quad (\boldsymbol{f})_{a_i} = \int_\Omega (\boldsymbol{P})_{iK}\, \varphi_{a,K}\, d\Omega - \int_\Omega \boldsymbol{b}\, \varphi_a\, d\Omega$$

where ρ is the material density and a_i denotes the i^{th} degree of freedom of the a^{th} node. We use a lumped mass matrix diagonalized by using a Lobatto quadrature. To solve for $\boldsymbol{\phi}^{k+1}$, we solve the nonlinear Eq. (4) via the Newton-Raphson iteration

$$\boldsymbol{\phi}^k_{p+1} = \boldsymbol{\phi}^k_p + [\nabla \boldsymbol{h}(\boldsymbol{\phi}^k_p)]^{-1}[\boldsymbol{p}^k - \boldsymbol{h}(\boldsymbol{\phi}^k_p)], \qquad p = 0, 1, \ldots, P$$

where the tangent matrix is given by

$$(\nabla \boldsymbol{h})_{a_i b_j} = \frac{1}{\Delta t}(\boldsymbol{m})_{a_i b_j} + \frac{\Delta t}{2}\int_\Omega (\partial_{\boldsymbol{\mathcal{F}}}\boldsymbol{P})_{iKjL}\, \varphi_{a,K}\, \varphi_{b,L}\, d\Omega \tag{6}$$

$$(\partial_{\boldsymbol{\mathcal{F}}}\boldsymbol{P})_{ijkl} = \mu\delta_{ik}\delta_{jl} + \lambda \boldsymbol{\mathcal{F}}^{-T}_{ij}\boldsymbol{\mathcal{F}}^{-T}_{kl} - (\lambda \ln(J) - \mu)\boldsymbol{\mathcal{F}}^{-T}_{li}\boldsymbol{\mathcal{F}}^{-T}_{jk} \tag{7}$$

Once $\boldsymbol{\phi}^{k+1} = \boldsymbol{\phi}^k_P$ has been computed within tolerance $\left\|\boldsymbol{p}^k - \boldsymbol{h}(\boldsymbol{\phi}^k_P)\right\| \leq \epsilon$, we determine $\boldsymbol{p}^{k+1}$ using Eq. (5). For more information on variational integrators, hyperelasticity, and neo-Hookean materials, see [20].

For simplicity, we use tetrahedral elements and affine basis functions. This makes computation of the deformation gradient particularly simple and compact. For each element,

$$\begin{aligned}\boldsymbol{\mathcal{F}}_{ij} = \frac{\partial \boldsymbol{\phi}_i}{\partial \boldsymbol{X}_j} &= \frac{\partial \boldsymbol{\phi}_i}{\partial \xi_k}\left[\frac{\partial \boldsymbol{X}_j}{\partial \xi_k}\right]^{-1} = \boldsymbol{\phi}_{a_i}\frac{\partial \varphi_a}{\partial \xi_k}\left[\boldsymbol{X}_{b_j}\frac{\partial \varphi_b}{\partial \xi_k}\right]^{-1} \\ &= \left[\boldsymbol{\phi}_1 - \boldsymbol{\phi}_4, \boldsymbol{\phi}_2 - \boldsymbol{\phi}_4, \boldsymbol{\phi}_3 - \boldsymbol{\phi}_4\right][\boldsymbol{X}_1 - \boldsymbol{X}_4, \boldsymbol{X}_2 - \boldsymbol{X}_4, \boldsymbol{X}_3 - \boldsymbol{X}_4]^{-1}\end{aligned}$$

where $\boldsymbol{X}_a$ is the position of node a in the reference configuration, and $\boldsymbol{\phi}_a = \phi(\boldsymbol{X}_a, t)$ is the position of the same node as a result of the deformation. The second matrix can be precomputed and stored as supplemental data to be passed to the element subroutine.

Thus, this problem requires multiple assemblies and sparse matrix solves per time step: one assembly and one sparse matrix solve per Newton-Raphson iteration. In the following sections we use the conjugate gradient method as our linear solver. Indeed, we will show that, with a GPU accelerated diagonally preconditioned conjugate gradient solver, a standard assembly stage run on the CPU is

the bottleneck. Thus, this problem can benefit greatly from a GPU accelerated assembly, which we provided in this chapter, to yield significant additional speed-up. As we show later, this and other time-integration schemes implemented in GPUs can yield stable, large time-step, real-time integrators for meshes of unprecedented size.

16.4.2 Implementation

The source code which accompanies this chapter is written in a modular fashion to accommodate testing and optimizing (see Figure 16.8). Each stage of the code has a CPU and a GPU version, which will perform the CPU-GPU transfers for the previous/subsequent stages as necessary, and all stages interface with a general sparse matrix class, allowing many sparse matrix formats to be tested easily.

The COO and CSR matrix structures are common sparse matrix formats and the HYB format from [8, 9] is a hybrid of the ELL and COO formats. The CSR_Vector and HYB matrix-vector product on the GPU both performed very well in tests in [8, 9] for matrices assembled from finite element models. Clearly though, this chapter's study should be extended to other formats, such as ELLPACK-R, PKT, DIA, and block-structured versions [7–9, 21].

16.4.2.1 Optimizations and Assumptions

The implemented matrices all have DXXX versions, which reserve the first portion of the matrix value array for the diagonal NZs of the matrix. For many sparse matrix formats and SpMV kernels, this

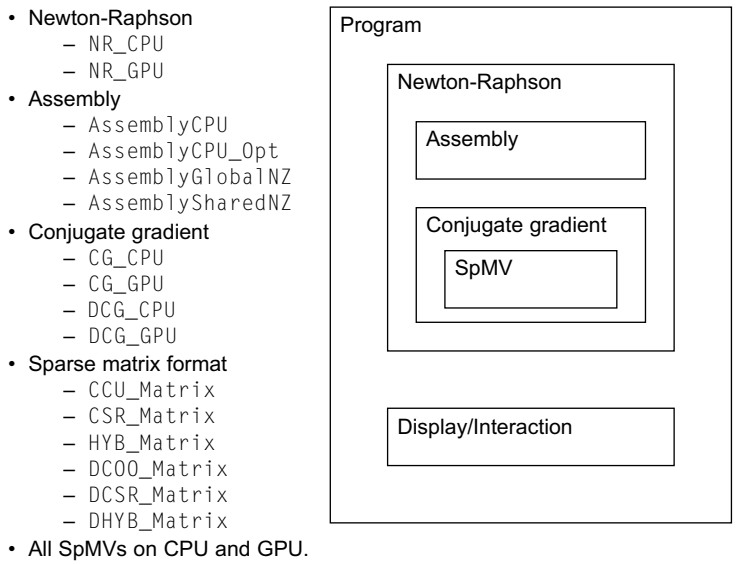


FIGURE 16.8

The structure of the program and some of the modular components available. `NR` denotes the Newton-Raphson iteration, `CG` denotes the conjugate gradient method, and `DCG` denotes the diagonally preconditioned conjugate gradient method. Multiple sparse matrix formats are available as well as their associated sparse matrix-vector multiplication (SpMV) implementations on the CPU and GPU.

is a small modification. This allows the diagonally preconditioned conjugate gradient method and the summation with the diagonal, lumped mass matrix in Eq. (6) to be implemented trivially and efficiently.

Rather than read in the reference configuration nodes for each assembly, we precompute the supplemental data

$$S(e) = \left[\boldsymbol{X}_1^e - \boldsymbol{X}_4^e, \boldsymbol{X}_2^e - \boldsymbol{X}_4^e, \boldsymbol{X}_3^e - \boldsymbol{X}_4^e\right]^{-1}$$

where $\boldsymbol{X}_a^e$, $a = 1,2,3,4$, denote the local reference configuration node for element e. Note that $\nabla \boldsymbol{h}^e$ and $\boldsymbol{f}^e$, the restriction of $\nabla \boldsymbol{h}$ and $\boldsymbol{f}$ to the domain of element e, are rotationally invariant when the material is isotropic, as in our construction. Thus, the supplemental data can be compressed from 9 values (3×3 matrix) to 6 values by performing an orientation-preserving QR decomposition and storing only the upper triangular part.

16.4.3 Validation

In this section, we validate the finite element assembly procedures by comparing the output of the single-precision GlobalNZ, SharedNZ, and a host implementation against a trivially implemented double-precision assembly on the host for a number of meshes. The validation is performed by assembling the tangent matrix $\nabla \boldsymbol{h}$ for a tetrahedral mesh of a unit sphere with various levels of refinement.

Table 16.1 lists the L^2 relative error between the entries of the system of equations constructed in double-precision on the host and single-precision on the device. As shown in Table 16.1, all of the device methods agree and exhibit the expected floating point accuracy, which increase with the inverse of the characteristic mesh size of the grid, $h \sim |\mathcal{N}|^{-1/3}$. This is expected since the entries of the system are functions of the distance between nodes, which have relative errors from the truncation of the nodal data proportional to $1/h$.

16.4.4 Performance Analysis

Our experimental setup is composed of an NVIDIA GeForce GTX 480 and Tesla C1060 processor installed on the PCI Express 2 bus (8 GB/s bandwidth) of a Intel Core 2 Quad CPU Q9450 2.66 GHz with 8 GB of RAM and running Linux kernel 2.6.32. The GTX 480 card has 15 multiprocessors, 480 cores, 1536 MB of memory, with a memory bandwidth of 177.4 GB per second. The C1060 card

Table 16.1 The L^2 Error of the Nonzeros in the System of Equations, $\left[\sum_{ij} |A_{ij}^s - A_{ij}^d|^2 / \sum_{ij} |A_{ij}^d|^2\right]^{1/2}$, between the Single-Precision Matrix, $\boldsymbol{A}^s$, and the Double-Precision Matrix, $\boldsymbol{A}^d$

		Sphere Mesh (Nodes, Elems)			
		63,120	**305,960**	**1.7K, 7.1K**	**3.6K, 16.5K**
Assembler and Matrix	CPU (float) COO	9.15e−8	1.05e−8	1.45e−7	1.77e−7
	SharedNZ CSR	8.75e−8	1.14e−8	1.45e−7	1.77e−7
	SharedNZ HYB	8.75e−8	1.14e−8	1.45e−7	1.77e−7
	GlobalNZ CSR	8.75e−8	1.14e−8	1.45e−7	1.77e−7
	GlobalNZ HYB	8.75e−8	1.14e−8	1.45e−7	1.77e−7

has 30 multiprocessors, 240 cores, 4 GB of memory, with a memory bandwidth of 102 GB per second. We use CUDA version 3.1, driver 256.35, `gcc` version 4.4.3, and `nvcc` release 3.1 version 0.2.1221.

Figure 16.9 shows the surface of a volumetric tetrahedral mesh consisting of 28,796 nodes and 125,127 elements representing a hand. (This mesh is available online at `aimatshape.net` and is commonly used to test graphics and physics applications.)

We now present benchmarks that illustrate that a GPU accelerated assembly provides a significant benefit for engineering applications, and is necessary if the goal is either graphics or real-time simulations. The GPU kernels are timed using CUDA events [22].

To begin, we run a system solely on the CPU using $\Delta t = 0.2$, $\mu = 5$, $\lambda = 2$, $\rho = 1$, and strict single-precision convergence – the Newton-Raphson tolerance is $\left\| \boldsymbol{p}^k - \boldsymbol{h}(\boldsymbol{\phi}_p^k) \right\|_2 < 10^{-5}$ and the conjugate gradient tolerance is $\|\boldsymbol{r}_p\|_2 < 10^{-6} \|\boldsymbol{r}_0\|_2$, where $\boldsymbol{r}_p$ is the residual at iteration p. The breakdown of the total time executing each stage is shown on the left of Figure 16.10. Clearly, at 70% of the total runtime, the SpMV is the bottleneck of this configuration and should be optimized. This stage can be optimized by implementing the conjugate gradient and sparse matrix-vector multiplication on the GPU and switching to the HYB matrix format, which is more efficient on the GPU. Running the same

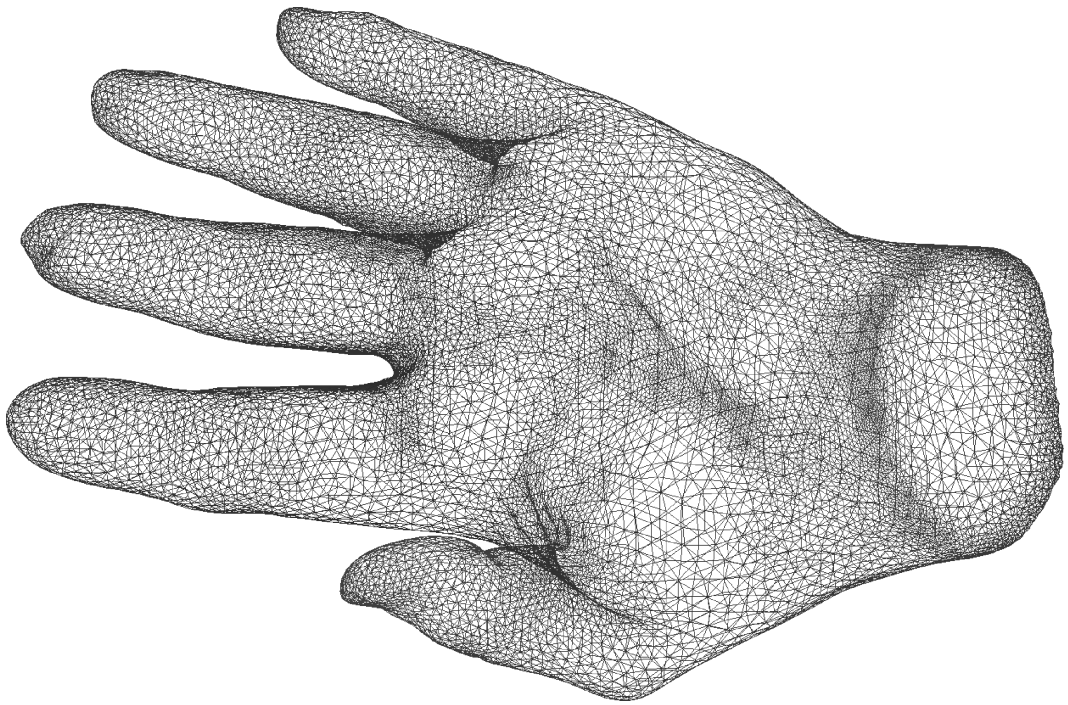


FIGURE 16.9

The surface of the tetrahedral mesh used in the validation and benchmarking sections. This mesh contains 28,796 nodes and 125,127 elements, resulting in a sparse linear system of size 86,388 and a matrix with 3.3M nonzeros.

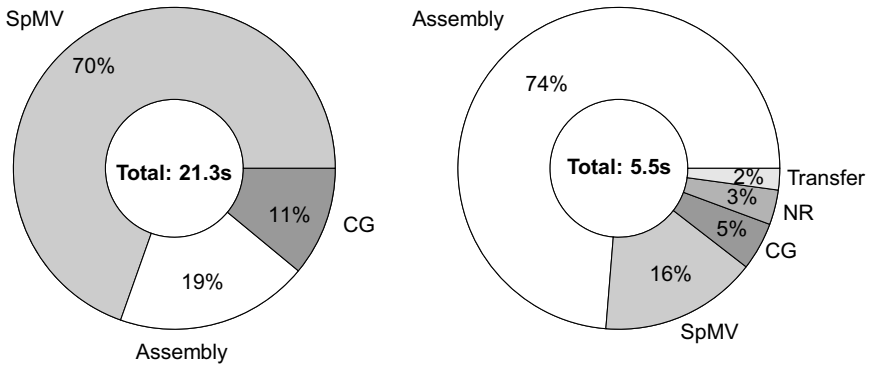


FIGURE 16.10

Optimizing only the conjugate gradient and SpMV for the GPU yields very good results, but ultimately has diminishing returns. (Left) All stages optimized and run on the CPU: `NR_CPU`, `AssemblyCPU_Opt`, `DCSR_Matrix`, `DCG_CPU`. (Right) Same run with SpMV and CG optimized and executed on the GPU: `NR_CPU`, `AssemblyCPU_Opt`, `DHYB_Matrix`, `DCG_GPU`.

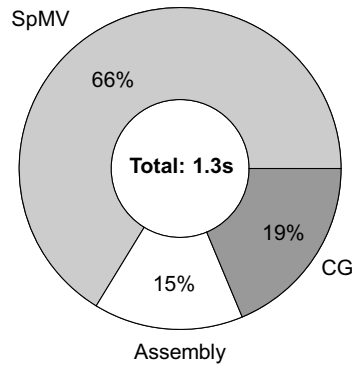


FIGURE 16.11

The breakdown of the total time spent in each stage when all components are optimized for and run on the GPU: `NR_GPU`, `AssemblyGlobalNZ`, `DHYB_Matrix`, `DCG_GPU`.

simulation with the optimized CG configuration, the new bottleneck is now the assembly on the CPU and the focus should no longer be on optimizing the SpMV on the GPU. Together with the 2% of time taken to perform the CPU-GPU transfers of data, the assembly now takes up 76% of the total runtime, while the entire conjugate gradient stage has been reduced to only 21%.

Using the methods in this chapter to implement the assembly on the GPU, a time profile similar to the pure CPU version is retrieved, as shown in Figure 16.11. The entire computation of the time step has been accelerated by over 16× from the CPU version. This is certainly beneficial for the engineering community, but by slightly relaxing the time step and convergence criteria, over 7 frames

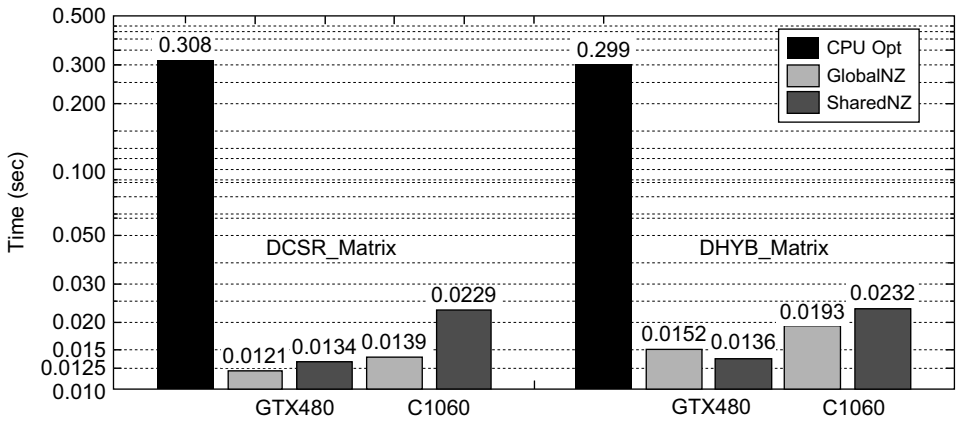


FIGURE 16.12

Comparison of the assembly operations with respect to hardware and matrix format. The `CPU_Opt` and `SharedNZ` assembly are not affected by the matrix format since they use direct indexing to write NZs. The `GlobalNZ` buffers the NZs in shared memory before writing coalesced blocks into the matrix (see color insert).

per second can be computed and rendered, allowing stable, real-time elastodynamics for meshes of unprecedented size. Figure 16.13 shows 4 frames from a real-time, interactive simulation which uses an OpenGL renderer and allows the user to dynamically apply forces to the mesh with the mouse.

Finally, in Figure 16.12 we compare the two approaches considered in this chapter with the optimized CPU assembly for both the DCSR and DHYB matrix formats. It is important to point out that, because the GlobalNZ reduction kernel buffers the NZs in shared memory before writing them in coalesced memory transaction to the matrix value array, it is affected by the DHYB's padded matrix format. Optimizing this stage to align with the matrix format's own alignment may be worthwhile. On the other hand, the SharedNZ assembly directly indexes into the matrix value array, sacrificing coalesced writes, but is not significantly affected by the matrix format. Additionally, the shared memory space on the GTX480 is expanded to 48K under the Fermi architecture, while the C1060 has only 16K per streaming multiprocessor, which disproportionately affects the performance of the SharedNZ method on the C1060.

16.4.5 Comparison with Previously Published Results

In the case of linear tetrahedral elastodynamics, each element has 4 nodes with 3 degrees of freedom each. Thus, the size of the elemental forcing vector $\boldsymbol{F}^e$ is 12 $(= 4 \cdot 3)$ and the size of the symmetric elemental matrix $\boldsymbol{A}^e$ is 78 $(= \frac{1}{2}4 \cdot 3(4 \cdot 3 + 1))$, so each element subroutine will compute 90 values.

In [1], the performance of the assembly routines were tested against the polynomial order of triangular elements, and therefore the size of the element data. Table IV of [1] shows that third order triangular elements compute 65 values and fourth order triangular elements compute 135 values. Thus, we may expect the performance of GlobalNZ and SharedNZ to be comparable to their third or fourth order performance values in [1]. However, the connectivity of the mesh plays an important role in the performance of these assembly algorithms. Specifically, nodes in the 2-dimensional mesh used in [1]

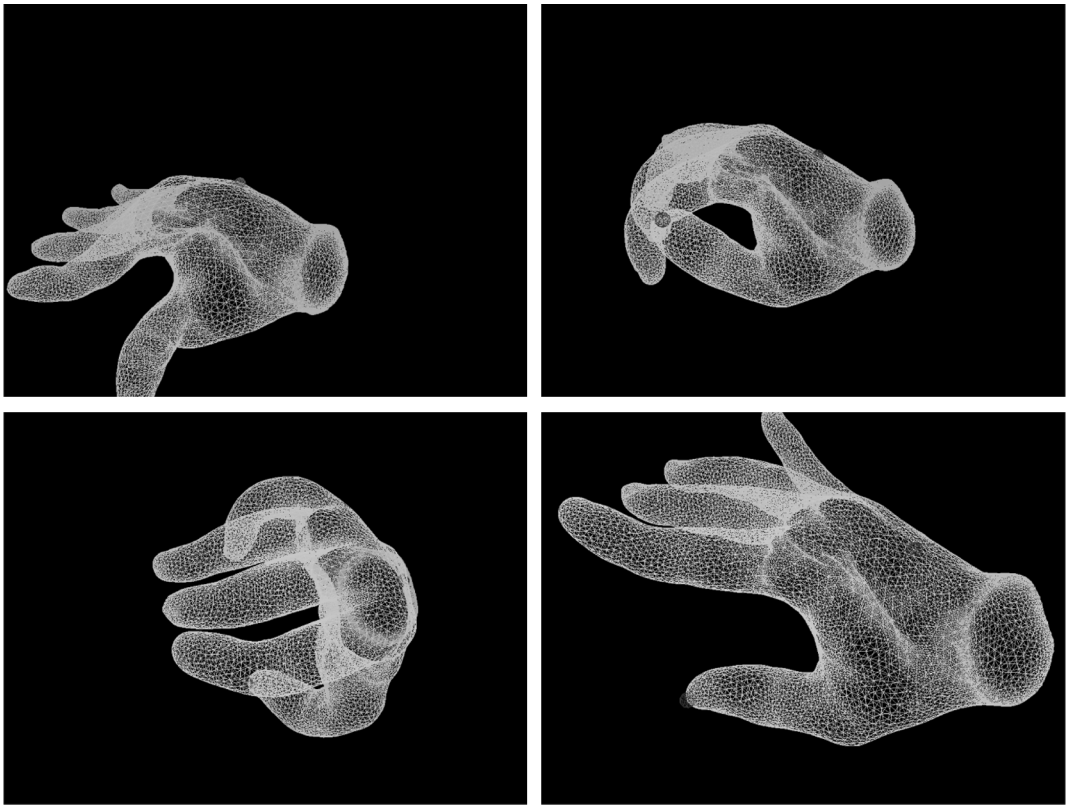


FIGURE 16.13

Sequence of images from elastodynamic simulation using the hand mesh with interactive user input forcing. The simulation is run at approximately 7–14 frames per second with $\Delta t = 0.01$, $\mu = 5$, $\lambda = 2$, $\rho = 1$, and single-precision convergence – the Newton-Raphson tolerance is $\left\| \boldsymbol{p}^k - \boldsymbol{h}(\boldsymbol{\phi}_p^k) \right\|_2 < 2 \cdot 10^{-5}$, and the conjugate gradient tolerance is $\|\boldsymbol{r}_p\|_2 < 10^{-4} \|\boldsymbol{r}_0\|_2$. Each time-step typically requires 3–6 NR iterations.

had, at most, only 9 adjacent triangular elements, whereas nodes in the 3-dimensional mesh shown in Figure 16.9 can have up to 42 adjacent tetrahedral elements. This seriously limits the size of the SharedNZ partitions, and is reflected in the relatively poor performance of SharedNZ on the C1060. On the GTX 480, this effect is reasonably offset by the expanded shared memory space, which offers a 48K shared memory space rather than the 16K on the C1060 and GTX 8800 used in [1].

16.5 FUTURE DIRECTIONS

There are a number of further optimizations that can be made in this problem, and which would likely have broader application.

First, also not considered in [1] is the recognition that, if n_f is the number of degrees of freedom per node and the degrees of freedom are numbered consecutively by node in the system of equations, then almost all NZs in the system appear in $n_f \times n_f$ blocks. These blocks can be assembled straightforwardly from the corresponding blocks in the element matrix, $\boldsymbol{A}^e$. This would compress the reduction array and allow for easier coalescing of data reads and writes. Similarly, in the SpMV, blocked versions of almost all matrix formats are possible and offer similar benefits.

Although the system of equations is symmetric, this is not taken advantage of in any of the methods in this chapter. Neither the assembly nor the SpMV use the symmetry as an optimization. Indeed, the authors are aware of only a few sparse matrix formats which are designed for symmetric matrices. Using an extra flag in the reductions arrays may allow an assembled NZ to be stored to two locations in the system of equations, potentially speeding up the reduction stage by almost a factor of two.

Although the code is templated to facilitate both single- and double-precision computing, more validation is needed before double-precision results can be included in this chapter. New generations of NVIDIA GPU hardware, such as the Tesla generation, which includes the GTX 480 used in this chapter, have greatly improved the architecture for computing with double-precision.

Testing double-precision versions of each stage on the GTX 480, with its new Fermi architecture designed with double-precision computing in mind, is an important new step. The code is written with templated classes to facilitate this. This is slightly more difficult to do on the GPU and more validation is needed before results can be included.

Acknowledgments

This work was partially supported by a research grant from the Academic Excellence Alliance program between King Abdullah University of Science and Technology and Stanford University. We also thank the Army High-Performance Computing and Research Center (AHPCRC) at Stanford for its support, as well as Juan-Pablo Samper-Mejia and Vivian Nguyen for their contribution during the 2010 AHPCRC Summer Institute.

References

[1] C. Cecka, A. Lew, E. Darve, Assembly of finite element methods on graphics processors, Int. J. Num. Meth. Eng. (2009).

[2] T.J.R. Hughes, The Finite Element Method: Linear Static and Dynamic Finite Element Analysis, Prentice-Hall, Englewood Cliffs, NJ, 1987.

[3] M. Rumpf, R. Strzodka, Graphics processor units: new prospects for parallel computing, in: A.M. Bruaset, A. Tveito (Eds.), Numerical Solution of Partial Differential Equations on Parallel Computers, vol. 51 of Lecture Notes in Computational Science and Engineering, Springer, 2005, pp. 89–134.

[4] D. Göddeke, R. Strzodka, J. Mohd-Yusof, P. McCormick, H. Wobker, C. Becker, et al., Using GPUs to improve multigrid solver performance on a cluster, Int. J. Comput. Sci. Eng. 4 (1) (2008) 36–55.

[5] R. Strzodka, D. Göddeke, Pipelined mixed precision algorithms on FPGAs for fast and accurate PDE solvers from low precision components, in: IEEE Symposium on Field-Programmable Custom Computing Machines 2006, 2006, pp. 259–268.

[6] D. Göddeke, R. Strzodka, S. Turek, Accelerating double precision FEM simulations with GPUs, in: Proceedings of ASIM 2005, 2005.

[7] L. Buatois, G. Caumon, B. Lévy, Concurrent number cruncher: an efficient sparse linear solver on the GPU, in: HPCC, 2007.
[8] N. Bell, M. Garland, Efficient sparse matrix-vector multiplication on CUDA, in: NVIDIA Technical Report, NVR-2008-004, 2008.
[9] N. Bell, M. Garland, Implementing sparse matrix-vector multiplication on throughput-oriented processors, in: Proceedings Supercomputing '09, 2009.
[10] M. Baskaran, R. Bordawekar, Optimizing sparse matrix-vector multiplication on GPUs using compile-time and run-time strategies, Technical Report, Research Report RC24704, IBM TJ Watson Research Center, 2008.
[11] J. Bolz, I. Farmer, E. Grinspun, P. Schröder, Sparse matrix solvers on the GPU: conjugate gradients and multigrid, ACM Trans. Graph. 22 (2003) 917–924.
[12] J. Rodriguez-Navarro, A. Susin, Non structured meshes for cloth GPU simulation using FEM, in: VRIPHYS, 2006, pp. 1–7.
[13] E. Tejada, T. Ertl, Large steps in GPU-based deformable bodies simulation, Simul. Model. Pract. Th. 13 (8) (2005) 703–715.
[14] E. Elsen, P. LeGresley, E. Darve, Large calculation of the flow over a hypersonic vehicle using a GPU, J. Comput. Phys. 227 (24) (2008) 10148–10161.
[15] D. Komatitsch, D. Micha, G. Erlebacher, Porting a high-order finite-element earthquake modeling application to NVIDIA graphics cards using CUDA, J. Parallel Distr. Com. 69 (5) (2009) 451–460.
[16] R.L. Graham, Bounds on multiprocessing timing anomalies, SIAM J. Appl. Math. 17 (1969) 263–269.
[17] G. Karypis, V. Kumar, MeTiS 4.0: Unstructured Graph Partitioning and Sparse Matrix Ordering System, 1998.
[18] J.E. Marsden, T.J.R. Hughes, Mathematical Foundations of Elasticity, Dover, Mineola, New York, 1994.
[19] A. Lew, J.E. Marsden, M. Ortiz, M. West, Variational time integrators, Int. J. Numer. Methods Eng. 60 (1) (2004) 153–212.
[20] J. Bonet, R.D. Wood, Nonlinear Continuum Mechanics for Finite Element Analysis, Cambridge University Press, 1997.
[21] F. Vázquez, E.M. Garzón, J.A. Martinez, J.J. Fernádez, The sparse matrix vector product on GPUs, Technical Report, University of Almeria, 2009.
[22] NVIDIA Corporation, NVIDIA CUDA Programming Guide 3.0. 2010.