INTERNATIONAL JOURNAL FOR NUMERICAL METHODS IN ENGINEERING
Int. J. Numer. Meth. Engng 2000; **00**:1–6 *Prepared using* nmeauth.cls *[Version: 2002/09/18 v2.02]*

Assembly of Finite Element Methods on Graphics Processors

Cris Cecka[1], Adrian J. Lew[1,2*], and E. Darve[1,2*]

[1] *Institute for Computational and Mathematical Engineering, Stanford University*
{ccecka,lewa,darve}@stanford.edu
[2] *Department of Mechanical Engineering, Stanford University*

SUMMARY

Recently, graphics processing units (GPUs) have had great success in accelerating many numerical computations. We present their application to computations on unstructured meshes such as those in finite element methods. Multiple approaches in assembling and solving sparse linear systems with NVIDIA GPUs and the Compute Unified Device Architecture (CUDA) are presented and discussed. Multiple strategies for efficient use of global, shared, and local memory, methods to achieve memory coalescing, and optimal choice of parameters are introduced. We find that with appropriate preprocessing and arrangement of support data, the GPU coprocessor achieves speedups of 30 or more in comparison to a well optimized serial implementation. We also find that the optimal assembly strategy depends on the order of polynomials used in the finite-element discretization. Copyright © 2000 John Wiley & Sons, Ltd.

KEY WORDS: finite element method, GPGPU, multicore, CUDA, high performance computing

1. Introduction

In the past several years, the emergence of general purpose computing on graphics processors (GPGPU) has sparked heightened interest in porting numerical algorithms to these new high-performance processors. GPUs are ideally suited to data-parallel computations with high arithmetic intensity. Applications taking advantage of this new technology have ranged from quantum chemistry [1] and molecular dynamics [2, 3] to fluid dynamics [4, 5] and cloth simulation [6]. GPGPU has the backing of hardware energized by the gaming market, is growing in popularity due to its substantial success, and is fueled by a move in the high-performance computing community towards heterogeneous system architectures where coprocessors are used to accelerate numerically intensive computations.

In the domain of partial differential equations, finite difference methods naturally fit into the GPU computing environment. Finite difference approaches have a regular data access

*Correspondence to: Mechanics and Computation, 496 Lomita Mall, Durand Building, Stanford University, Stanford, California 94305-4040, U.S.A.

Contract/grant sponsor: ONR Young Investigator Award; contract/grant number: N000140810852

Received

Copyright © 2000 John Wiley & Sons, Ltd.

Notation	Description
n_d	spacial dimension
n_f	degrees of freedom per node
e_n	number of nodes per element
e_f	degrees of freedom per element, $e_n n_f$
$\mathcal{N}$	set of mesh nodes
$\mathcal{E}$	set of mesh elements
$C(n)$	nodal data matrix. Section 3.1
$E(e, a)$	connectivity matrix. Section 3.1
$L(e, d)$	location matrix. Section 3.2

Table I. Table of notations

Term	Explanation
Host	The CPU
Device	The GPU
Nodal Data	The coordinates and nodal values of any fields required by an element subroutine.
Element Data	The elemental matrix $\boldsymbol{A}^{\mathrm{e}}$ and forcing vector $\boldsymbol{F}^{\mathrm{e}}$ output by an element subroutine.
Kernel	A function executed in parallel on the device.
Thread	A process on the device executing a kernel.
Block/Thread Block	A set of threads which share a multiprocessor, its shared memory space, and thread synchronization primitives.
Warp	16 or 32 threads executing synchronously within a block.
Grid	A set of blocks executing a single kernel.
Local Memory	Any device memory exclusive to a single thread.
Shared Memory	On chip memory that can be shared among threads of a single block.
Global Memory	Off chip memory that is accessible to all threads.
Coalesced Memory Access	Multiple global memory accesses that are conglomerated into a single memory transaction by the device. Requires appropriate memory access alignment and contiguity.

Table II. Glossary of terms

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using nmeauth.cls

pattern and do not require storing mesh connectivity information, making them a natural candidate for execution on GPUs. When solving partial differential equations in complex domains or when adaptive mesh refinement strategies are needed, unstructured grids are often convenient to use. Computations on an unstructured grid follow a typical pattern in which data is first gathered, calculations are performed, and the resulting data is reduced and scattered. For example, in a finite-element computation, each element requires nodal data, computes element contributions to the stiffness matrix and forcing vectors, and finally assembles these into the linear system of equations (reduction step). Two key difficulties result from such an algorithm: the amount of data to be read and written to global memory is typically quite large in comparison with the amount of floating point operations to perform, and the reduction step is made difficult by the concurrent execution of many threads on the GPU. These two features suggest that unstructured grid calculations are not well suited to GPUs, at least when compared to calculations on a structured grid. However, we show in this paper that in fact excellent performance can still be extracted from the hardware.

In this paper, we investigate the use of a single GPU to accelerate the assembly and solution of general finite element methods (FEMs) on unstructured meshes. We leave the problem of running on multiple GPUs for future study. We discuss approaches that are fairly general and relevant to many types of computations on unstructured meshes. We explore and classify a range of possible approaches, report on our experiences, and make recommendations to potential implementers. The more successful approaches are compared using a two-dimensional benchmark case. All the methods and optimizations presented in this paper can be extended almost unchanged to a three-dimensional model. However, the conclusions about what constitutes an optimal strategy may change in this case.

Previous studies on FEM in GPUs have largely focused on the solution of the sparse linear system of equations resulting from a FEM discretization [7, 8, 9], mainly because this is often the most computation-intensive step. Some assembly strategies for the GPU have been discussed as well. Often, specific applications have allowed special approaches for FEM assembly. The methods described in [9] for geometric flow on an unstructured mesh and in [6] for FEM cloth simulation derive relatively simple expressions for each non-zero in the system of equations. This relative simplicity and the fact that each entry can be computed using a small amount of input is well suited to GPUs. In this paper, we attempt to consider methods for finite element assembly that are general enough to be used for a wide range of finite element models.

The implementation of discontinuous Galerkin methods on the GPU was considered in [10] and applied to Maxwell's equations. For high-order, continuous Galerkin methods an assembly strategy is proposed in [11]. A coloring of the elements was used in the matrix assembly for parallel execution. A version of this approach is presented in this paper, although we found that it does not perform as well as other methods in our benchmarks.

Other successful approaches such as in [12] for deformable body simulation took advantage of vertex and fragment shaders on the GPU to perform the assembly. New hardware and a more general computing environment appear to have made some of these techniques obsolete. However, the approach is still relevant and can be thought of as the Global Assembly by NZ method considered in this paper. Algorithms developed for other shared memory machines such as presented in [13] have been considered and extended for this paper.

The paper is organized as follows. First, we introduce NVIDIA's CUDA and GPU architecture in Section 2. The key difference between a "conventional" processor and a GPU is

Copyright © 2000 John Wiley & Sons, Ltd.
Prepared using nmeauth.cls

the fact that GPUs have hundreds of cores and can process thousands of threads concurrently. In addition, in most cases, the cores on the processor must execute the same instruction at the same time. This architecture is sometimes described as a Single Instruction Multiple Thread architecture. NVIDIA GPUs are also characterized by the different types of memories that can be explicitly accessed by the programmer. In particular, some processors have an on-chip memory that can be shared by multiple threads. This is a key component to optimize the code and reduce the movement of data in and out of the off-chip memory, and improve performance in many cases.

In Section 3, we discuss basic features of finite element methods on unstructured grids, and review the most standard data structures and assembly procedure for serial implementations. In Section 3.4, we introduce considerations for efficiently implementing the assembly on streaming processors and, specifically, on NVIDIA GPUs, and discuss a range of possible approaches. In the basic approach that mimics the serial algorithm, threads compute data for an element and then scatter the results to the matrix and forcing vector of the linear system of equations. We also considered algorithms in which threads are primarily in charge of computing a single non-zero entry or an entire row in the matrix. We discuss the pros and cons of each technique and in particular how they can efficiently make use of the different hardware resources, such as the off-chip memory and the on-chip memory shared by multiple threads.

In Section 4, we detail and analyze the most successful techniques, and explain the different optimizations that are required to improve performance. Section 5 discusses the main approaches for solving the resulting system of equations. The proposed methods for FEM assembly are validated in Section 6.1 and relative running times are presented in Section 6.3. An NVIDIA GeForce 8800 GTX, an NVIDIA Tesla C1060, and a single core of an Intel Core 2 Quad CPU Q9450 2.66 GHz, were used in the measurements. The performance is analyzed as a function of the problem size (number of mesh nodes), and the order of the finite-element method.

2. GPU Architecture and CUDA

2.1. GPU Architecture

The NVIDIA GPU architecture is composed of some number of *streaming multiprocessors* (SMs) each containing a number of *streaming processor cores* (SPs) and a single instruction unit. At the time of this writing, the SPs are capable of performing one integer or single precision floating point operation per clock cycle. On a modern GPU, a single floating point operation may be an addition, a multiplication, or a multiply-add. Many devices also include a shared unit for double precision floating point operations. All SMs have access to *global memory*, the off-chip memory (DRAM) which has a latency of several hundred clock cycles. Each SM has on-chip memory which is partitioned into *register spaces* for *threads* executed on the SPs. The SPs within an SM may communicate through banked *shared memory*, another on-chip memory with latency comparable to register memory [14].

The programmer has a fine level of control over the different memories and caches, allowing several optimizations. The memories available on NVIDIA GPUs are

1. On-chip registers

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using nmeauth.cls

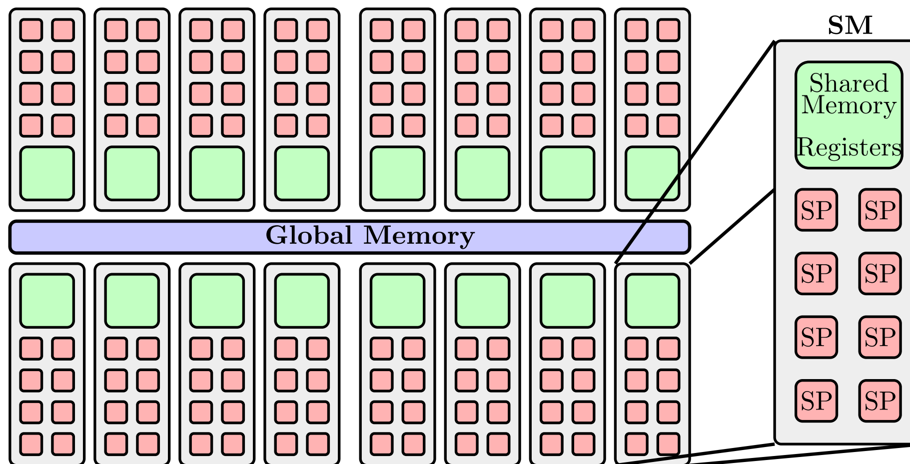


Figure 1. Diagram of the 8800 GTX with 16 SMs and 128 SPs. Each SM has shared and register memory and all have access to the off-chip global memory.

2. On-chip shared memory
3. Local memory
4. Off-chip global memory
5. Off-chip texture memory with two-level caching
6. Off-chip constant memory with two-level caching

The term local memory is used to describe memory local to each thread. This memory can be physically allocated on-chip or off-chip, when the on-chip space is depleted.

The NVIDIA GeForce 8800 GTX has 16 SMs consisting of 8 SPs each with a 575MHz core clock. It has 768MB of global memory with a memory bandwidth of 86.4GB/sec, see Fig. 1. The two-level caching is believed to be a fully associative 16-entry translation lookaside buffer (TLB), a 5KB 20-way set associative L1 cache with 160 cache lines in 8 fully associative sets and latency approximately half of the DRAM, and a 32KB 24-way set-associative L2 cache [15].

The NVIDIA Tesla C1060 has 30 SMs consisting of 8 SPs each with a 1.3GHz core clock. It has 4GB of global memory with a memory bandwidth of 102GB/sec.

2.2. Execution Model

NVIDIA's Compute Unified Device Architecture (CUDA) parallel programming model provides a C/C++ language interface to the hardware [14] (as well as a recent FORTRAN port by the Portland group). A CUDA application consists of a sequential *host* program run on the CPU that launches *kernels* written in CUDA to be run on the parallel GPU *device*. A kernel is a Single Instruction Multiple Thread (SIMT) computation that is executed in parallel by a set of threads. Threads are grouped in *blocks* and a kernel runs a *grid* of one or more thread blocks. Each block is run by a single SM and threads in the same block may communicate via the shared memory and synchronization primitives in CUDA. Blocks of a grid do not share data and cannot be synchronized. Additionally, each thread has a private local memory and

Copyright © 2000 John Wiley & Sons, Ltd.

register space. Off-chip global memory is accessible by all threads.

Threads are executed on the SPs in groups called *warps*, the smallest scheduling unit. The SPs share a single instruction unit in the SM so all threads of a warp execute the same instruction at the same time. The CUDA runtime scheduler is a transparent warp scheduler on hardware with no overhead. It allows threads within a warp to take different branches in the kernel by executing the branches in sequence. This distinguishes the architecture from Single Instruction Multiple Data (SIMD) architectures.

2.3. Optimizing the GPU

Although all threads have full access to global memory, referencing scattered locations causes the hardware to serialize the memory transactions. However, if the global memory is accessed by consecutive threads in consecutive locations and with the correct alignment then the memory transactions are *coalesced* into one single transaction. The specific requirements to achieve memory coalescing varies across GPUs and are becoming less restrictive in more recent GPU generations. See [14] for details.

Since the shared memory is banked, if all threads access a different bank the transaction is completely parallel. Additionally, if all threads access the same memory location, the bank issues a broadcast transaction that requires the same number of clock cycles. Otherwise, the transaction is performed sequentially in conflict free maximal subsets.

As a rule-of-thumb, the fastest kernels minimize global memory transactions, avoid shared memory bank conflicts, and minimize register and shared memory usage to maximize thread occupancy. At the same time, one strives to work with many blocks running per multiprocessor to overlap the latencies of memory transfers with useful computation.

3. Computations on Unstructured Meshes

This paper focuses on the assembly of a finite element system over unstructured meshes using the massively parallel capabilities of NVIDIA GPUs. We describe the essential ideas behind the algorithms we propose in this part of the paper. To this end, we introduce elementary notions on finite elements in Section 3.1, aimed at the unfamiliar reader. In Section 3.2, a typical finite element assembly procedure is reviewed and the important primitive operations and data flow discussed. We comment on some aspects of the implementation of essential boundary conditions in Section 3.3. In Section 3.4, we present optimization guidelines and the possible strategies we considered in creating a finite element assembly procedure on a recent GPU architecture. After presenting the merits and pitfalls of each approach, we present an outline of our most successful optimized methods in Section 4.

3.1. Finite Element Method

An unstructured mesh over a polygonal (polyhedral in 3D) domain Ω is a partition of the domain into disjoint polygons (polyhedra) $\mathcal{E}$, termed elements, such that $\Omega = \overline{\bigcup_{e\in\mathcal{E}} \Omega^e}$, where Ω^e is the set of points in element $e \in \mathcal{E}$. Unstructured meshes are common in many engineering and graphics applications and are used to create versatile discretizations of partial differential equations (PDEs). To illustrate this, consider the problem of finding a function $u\colon \Omega \to \mathbb{R}$ that

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using nmeauth.cls

satisfies

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

and subject to some boundary conditions on the boundary of Ω, $\partial\Omega$. Here Ω is a domain in $\mathbb{R}^{n_d}$, $\mathcal{L}$ is a general linear differential operator, and f is a scalar-valued function over Ω. Equilibrium stresses and temperatures are typically modelled by PDEs such as (1).

A standard finite element method to approximate this problem begins by constructing a finite dimensional space $\mathcal{V}_h$ of functions over Ω. Finite element procedures facilitate the construction of a basis $\{\varphi_i\}$ for $\mathcal{V}_h$. The numerical approximation of the solution can then be written as

$$u_h(\boldsymbol{x}) = \sum_j u_j \varphi_j(\boldsymbol{x}), \tag{2}$$

where $\boldsymbol{u} = [u_i]$ are the components of u_h in basis $\{\varphi_i\}$. A typical formulation of a finite element method can then be obtained by application of Galerkin's method. In this case u_h is such that

$$a(u_h, \varphi_i) = \int_\Omega f \varphi_i \, d\Omega, \quad \forall \varphi_i, \tag{3}$$

where $a \colon \mathcal{V}_h^2 \to \mathbb{R}$ is defined as

$$a(u_h, \varphi_i) = \sum_j u_j \int_\Omega \mathcal{L}\varphi_j \; \varphi_i \, d\Omega. \tag{4}$$

Very often the basis functions $\{\varphi_j\}$ are not smooth enough for $\mathcal{L}\varphi_i$ to be a function. It is typically a distribution. This can be circumvented by appropriately integrating by parts the second term on the right hand side of (3). Equation (3) can be alternatively expressed as a linear system of equations of the form

$$\boldsymbol{A}\boldsymbol{u} = \boldsymbol{F}, \tag{5}$$

where $\boldsymbol{A}$ is the stiffness matrix and $\boldsymbol{F}$ is the forcing vector. The numerical approximation u_h follows by solving this system for the vector of components $\boldsymbol{u}$.

For some finite element spaces, it is possible to choose points $\boldsymbol{x}_i \in \Omega$ termed nodes such that $\varphi_j(\boldsymbol{x}_i) = \delta_{ij}$. For these spaces, the values of u_h at all nodes uniquely determine it. By the way basis functions are constructed, we have that whenever Ω^e is included in the support of φ_i, then $\boldsymbol{x}_i \in \overline{\Omega^e}$. For example, if node $\boldsymbol{x}_i$ is in the interior of an element, then φ_i is non-zero only in that element. Similarly, when $\boldsymbol{x}_i$ belongs to an edge shared between two elements, φ_i is non-zero in these two elements only, and so on. Therefore, this property means that finite element basis functions $\{\varphi_j\}$ have their support restricted to at most a few neighboring elements. Consequently, the number of nodes that belong to any given element does not change as the number of elements in the mesh is increased, and it is generally much smaller than the total number of nodes in the mesh.

Many variants to this basic structure exists. For example, for problems in which the solution is a vector field, such as the displacement field in elasticity, it is possible to approximate each Cartesian component of the vector field with (2). In this case there is more than one unknown or degree of freedom per node of the mesh, and the matrix $\boldsymbol{A}$ and forcing vector $\boldsymbol{F}$ are therefore larger. Some finite element spaces are specifically built for vector fields, such as the Brezzi-Douglas-Marini family [16]. More important changes are found in problems involving multiple

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using nmeauth.cls

fields that use different basis functions, such as for incompressible flows. Many of the things discussed in this paper apply to some of these cases as well, but we do not address these directly here. In this paper we restrict the discussion to finite element methods in which each node is associated to a basis function in the way described earlier, with exactly one degree of freedom per node.

Two 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 nth node, with n referred to here as the global node number. The first fields are often the coordinates of the node, followed by nodal values of any 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 all degrees of freedom, such as temperature or displacement.
2. The connectivity matrix $E(e, a)$, which yields the global node number of the ath node of the eth element, with a referred to here as the local node number.

Usually, the computation of the numerical solution involves two steps: the assembly of matrix $\boldsymbol{A}$ and vector $\boldsymbol{F}$, and the solution of the linear system for $\boldsymbol{u}$. Many applications often require the assembly and solution of many of these types of systems. This is the case when $\mathcal{L}$ is a non-linear differential operator, for which an iterative strategy, such as a (modified) Newton-Raphson, is often adopted to find $\boldsymbol{u}$. Since the matrix $\boldsymbol{A}$ depends on the value of each candidate solution $\boldsymbol{u}$, it needs to be assembled several times to solve the problem. Problems in which a parametric dependence of the solution $\boldsymbol{u}$ needs to be computed, such as time-dependent solutions, only exacerbate the importance of a fast assembly strategy like the ones explored here.

3.2. Sequential FEM Assembly

The assembly procedure partitions the integrals over Ω in (3) 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. For example, the bilinear form defined in (3) 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 (4) to Ω^e.

The computation of

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

on element e is different than zero if and only if Ω^e is included in the supports of both φ_i and φ_j, i.e., if nodes at which they are identically 1 belong to e. Consequently, during assembly A^e_{ij} needs to be computed for only a few values of i and j on each element. These values need then be accumulated into matrix $\boldsymbol{A}$.

A typical finite element assembly program relies on given *element subroutines* to compute an element matrix $\boldsymbol{A}^{\mathrm{e}}$ and element forcing vector $\boldsymbol{F}^{\mathrm{e}}$. These element subroutines change with the PDE, the element type, and the basis functions, and are functions of the nodal coordinates, and any nodal fields, forces, or boundary conditions. The input arguments are the nodal data contained in $C(n)$, for each global node number n in the element. These values are generally retrieved from memory and arranged following a local node numbering scheme for the element.

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using nmeauth.cls

Similarly, after the element data $\boldsymbol{A}^e$ and $\boldsymbol{F}^e$ are computed, these data are accumulated in $\boldsymbol{A}$ and $\boldsymbol{F}$ following the map from local to global degrees of freedom.

This mapping information is stored in the *location matrix* L. For the dth degree of freedom of the eth 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

$$\boldsymbol{A}(i,j) = \sum_{\substack{e,d_1,d_2\\ L(e,d_1)=i\\ L(e,d_2)=j}} \boldsymbol{A}^{\mathrm{e}}(d_1,d_2) \qquad\qquad \boldsymbol{F}(i) = \sum_{\substack{e,d\\ L(e,d)=i}} \boldsymbol{F}^{\mathrm{e}}(d), \tag{6}$$

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.

```
1 Initialize A and F to zero;
2 for all elements e ∈ E do
3   (A^e, F^e) ← elem(e) ;                       /* elemental subroutine */
4   for all local degrees of freedom d1 of e do
5     F(L(e, d1)) += F^e(d1);
6     for all local degrees of freedom d2 of e do
7       A(L(e, d1), L(e, d2)) += A^e(d1, d2);
```

Algorithm 1: The direct stiffness method of finite element assembly.

This is known as the direct stiffness method and is the most common implementation of a system assembly in the finite element method.

3.3. On Essential Boundary Conditions

Essential boundary conditions constrain the solution u of (1) to satisfy

$$u = g \quad \text{on} \quad \Gamma_g \subset \partial\Omega, \tag{7}$$

for some prescribed boundary value g. For the most common finite element formulations this translates into imposing the nodal values of u_h at the nodes on Γ_g. The imposed nodal values can be computed in a variety of ways, the most of common of which is pointwise evaluation of g at the nodes.

Two types of implementation are often encountered. The first consists in assembling $\boldsymbol{A}$ and $\boldsymbol{F}$ with the prescribed degrees of freedom as unknowns, like any other degree of freedom. Then, $\boldsymbol{A}$ and $\boldsymbol{F}$ are modified by zeroing rows and columns associated with the prescribed degrees of freedom, adding a one to the corresponding diagonal entry, and appropriately modifying the right hand side. The second approach is to not include the prescribed degrees of freedom as unknowns in the linear system of equations. In this case the elemental subroutine does not assemble the prescribed degrees of freedom and appropriately modifies the computation of the right hand side, as shown in Section 6.2.

For our benchmark examples we have adopted the second approach. For clarity, however, the discussion of the assembly algorithms is done for the first approach, in which all nodes in an element are dealt with identically. Implementing the second approach involves passing

Copyright © 2000 John Wiley & Sons, Ltd.

additional arguments to the elemental subroutine indicating which degrees of freedom to compute, what the prescribed values are, if any, and what terms of the right hand side to modify.

3.4. CUDA FEM Assembly

The key concerns in mapping a numerical method to an algorithm suitable to run on a GPU include:

1. Decomposing the task into independent blocks of work. No inter-block communication is available in the GPU execution model.
2. Determining a data flow that minimizes global memory transactions, conform to the device's coalesced memory transaction requirements, and takes advantage of the exposed memory hierarchy to improve data reuse and access efficiency.
3. Balancing the amount of (possibly redundant) computation with the efficiency of the data flow.
4. Optimize the number of threads per block and the number of blocks that can run on an SM concurrently.

The primary difficulty in adapting the direct stiffness method to the CUDA GPU architecture is efficiently moving nodal data to the element kernels and then accumulating the elemental data $\boldsymbol{A}^e$ and $\boldsymbol{F}^e$ into the system of equations. Because the mesh is unstructured, reading and writing coalesced global memory and avoiding shared memory bank conflicts may be impossible without significant and time-consuming restructuring of the input data. Furthermore, finite element codes always have degrees of freedom shared among multiple mesh elements, and because there are no global synchronization primitives and atomic operations are either not available or undesirable, care must be taken to avoid race conditions.

A race condition arises when the order of execution of the code unintentionally affects the output. In our case, this may happen when at least two threads accumulate numbers at the same memory location. Suppose these threads read a memory location, accumulate locally some data and then write back. A race condition occurs when we cannot guarantee that one threads is able to read and write before another thread reads the same memory location.

As an example, Bolz et. al. in [9] briefly mention the assembly of a sparse matrix in an FEM context. Their recommendation is a natural approach that calls for each thread to take responsibility for computing one non-zero entry (NZ) in the system of equations $\boldsymbol{A}\boldsymbol{u} = \boldsymbol{F}$. This immediately prevents any possible race conditions. However, generalizing this approach requires that each thread runs the element subroutine for each contributing element and extracts the relevant entry from each element matrix and element forcing vector. Consequently, this method results in a high degree of redundant computation. If e_f is the number of degrees of freedom per element, then the element subroutine will be called $(e_f + 1)e_f$ times – one for each NZ in a nonsymmetric $\boldsymbol{A}^{\mathrm{e}}$ and one for each NZ in $\boldsymbol{F}^{\mathrm{e}}$. Even if the computation in the element subroutine is inexpensive, retrieving the nodal data required by the element subroutine multiple times from the global memory without being able to take advantage of coalescing is expensive.

In this paper, we consider the element subroutine as a black-box to be executed in its entirety by each thread. This may be the case for a wide range of low-order finite element methods that are ideal for acceleration using a GPU. Problem-dependent optimizations like those found

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using nmeauth.cls

in [9, 12, 4] and high-order optimizations such as parallelizing the element subroutine [11] can also be investigated for further algorithm improvement.

We considered and implemented multiple approaches to this problem. We briefly summarize a number of choices that were made. First, we must decide what to associate threads with. Some natural choices include making threads primarily associated, at a time, with an element, with a non-zero entry of the matrix, or with a matrix row or other subsets of non-zero entries. Each option carries its own implications, summarized below.

1. Assembly by NZ: One thread is responsible for one NZ of the system of equations at a time. This choice removes dependence on the sparsity pattern and storage type of the sparse matrix as it only operates on a set of indices into $\boldsymbol{A}$ and $\boldsymbol{F}$ which are passed to it. However, NZs of the system of equations may have significantly different computational requirements. For example, NZs on the diagonal of the system of equations typically have many elements contributing to them while off-diagonal NZs may have only one or two. This imbalance causes problems in certain implementations.
2. Assembly by Row: One thread is responsible for one row of the system of equations at a time. The computational requirements are much more constant by row than by NZ. However, an extra search or lookup must be performed to find the correct column in the row whenever updating a value. Using a search may cause thread divergence (branching). Additionally, dependence on the storage format of the system in memory may cause problems.
3. Assembly by Element: One thread is responsible for computing and assembling the entries for one element of the mesh at a time. This approach is most similar to the serial direct stiffness assembly method as an element is computed and then immediately accumulated into the system of equations. Each thread performs the same amount of computation. The entries to be modified in the system of equations can be determined a priori for each element, preventing any dependence on the storage format of the system. In contrast to the previous two strategies, parallel versions must take special care to avoid race conditions when updating the system of equations. A common approach is to color the elements of the mesh such that elements with the same color do not share degrees of freedom. The parallel method is then run for each color sequentially.

Secondly, we must decide how to use the hardware resources available to us. The latest NVIDIA GPUs provide three primary memory levels: global memory available to all threads with scope of the application, shared memory available to all threads of a single block with scope of the kernel, and local memory available to a single thread with scope of the thread. Primarily, we must decide where to store the element data. Again, each option carries implications for the resulting algorithm, summarized below.

1. Global: All element data are stored in global memory. This allows the element data computation stage to be distinct from the assembly operation as one kernel can be responsible for computing the element data in global memory and another kernel will assemble that data. Thus, in the computation kernel threads will be associated with elements. In the assembly kernel threads will be associated with either elements, rows, or NZs. Since each element is computed only once, this approach is promising if the element subroutines are very expensive. Storing element data in global memory requires large amounts of memory to be allocated. The computation can be optimized with coalesced

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using nmeauth.cls

writing of the element data and can use less memory by partitioning the elements and performing multiple computation-assembly passes for a single assembly of the system of equations. Efficient reduction into the system of equations may be difficult since the element data resides in global memory.

2. Local: The element data are stored in local memory. Since local memory is not shared between threads, if the thread is associated with a row or NZ, the element subroutines must be run as needed. Thus, almost every element will be computed more than once. In the case of assembling by element with coloring, no sharing of element data or nodal data is expected, and so using local memory may be optimal.
3. Shared: The element data are stored in shared memory. Since shared memory is accessible by a block of threads, multiple threads may use the result to assemble rows or NZs. Thus, in the first stage of the kernel threads are associated with elements. In the second stage, threads are associated with rows or NZs and assemble from shared memory. Due to the limited amount of shared memory, the number of elements that may be stored at once is often relatively small. The shared memory latency is comparable to register memory latency making this approach potentially very efficient.

These two sets of choices are mostly independent, although some combinations make more sense than others. For example, Global-Element would imply the following algorithm

- Associate each computation thread with an element.
- Store the element data into global memory.
- Associate each assembly thread with an element.
- Read elements and assemble into the system of equations.

which can be optimized by neglecting the store-read step. On the other hand, Local-Element looks like

- Associate each computation thread with an element.
- Store the element data in local memory.
- Assemble into the system of equations.

which appears to be a good algorithm. These can be summarized in Table III.

	Element Data		
Assemble By (Associate Threads With)	Global	Local	Shared
NZ	Precompute all element data into global memory and assemble by NZ.	Assemble by NZ, computing element data as needed in local memory.	Store multiple elements in shared memory and assemble by NZ.
Row	Precompute all element data into global memory and assemble by row.	Assemble by row, computing element data as needed in local memory.	Store multiple elements in shared memory and assemble by row.
Element	Precompute all element data into global memory and assemble by element.	Compute and store element data in local memory and assemble into the system.	Store element data in shared memory and assemble into the system.

Table III. A table of approaches considered in this paper.

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using nmeauth.cls

The method described in [9] corresponds to the assembly by NZ in local memory. The method described in [12] corresponds to the assembly by row from global memory. The method described in [11] corresponds to the assembly by element using a disjoint coloring of mesh elements and local memory.

In the following sections, we will describe the main approaches we have considered and how they were optimized. The first one, in Section 4.1, uses the idea of mesh coloring (see [11]). In this approach, each thread is responsible to perform operations for one element and then accumulates the result into the system of equations. The coloring prevents two threads from writing to the same memory location. The advantage of this method is its relative simplicity, and the fact that there are no redundant operations. However, in our tests the running time of this algorithm is not competitive due to the excessive number of global memory transactions.

The second method, in Section 4.2, consists of decomposing the calculation into two phases: we calculate all the element data and write it back to global memory. Then we assemble all the element data into the matrix $\boldsymbol{A}$. The advantage of this method is that it does not require a lot of shared memory or register space. Consequently, it is well suited for high order methods. We observed that indeed it has the smallest running time for methods of order 4 and above (see Section 6.3).

The third method, in Section 4.3, is the most complex and uses the shared memory to reduce the amount of data read and written to global memory. The nodes and elements are partitioned into subdomains. For each subdomain, threads assemble all the element data and then calculate the corresponding entries in the matrix $\boldsymbol{A}$. This reduces the number of times nodal data are read, avoids an excessive recomputation of the element data, and also reduces the number of global memory writes. Consequently, for methods of order less or equal to three, this is the fastest. However it depends heavily on shared memory space and becomes slower for higher order methods and, in our case, fails altogether for order five.

Finally in Section 4.4, we discuss a technique that uses local memory. In this approach, each thread is responsible for one entry in the matrix and performs all the calculation required for that entry. We included this method for completeness. However, the amount of recomputation involved makes this method non-competitive.

These techniques are now described in more details.

4. Algorithms for Finite Element Assembly on GPUs

4.1. Assembly by Element via Coloring

One technique to avoid a race condition in the direct assembly method is to precompute a coloring of the mesh elements such that no two elements of the same color share any given degree of freedom, see Fig. 2 for an example. Then, it is safe to run a parallel version of the direct assembly method one color at a time. A similar approach was taken by Komatitsch et al. in [11]. The main difference is that multiple threads were used to compute the high-order element data for a single element. Specifically, a thread block of 128 threads was assigned to each 125-node spectral element.

4.1.1. Coloring the Mesh Elements. Although determining the minimum coloring of a general graph is known to be an NP-complete problem, there exist many heuristics for finding nearly

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using nmeauth.cls

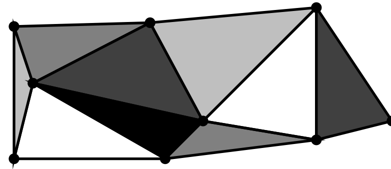


Figure 2. A 2D colored triangular element mesh.

minimal colorings [17]. In this application, we wish to find a vertex coloring of the dual graph of the mesh, in which each element is a node of the graph and two nodes are connected by an edge if the corresponding elements of the primal mesh share a node. This can be computed in polynomial time if the dual graph is chordal, which is the case in a number of FEM applications. It should be noted that many graph coloring algorithms (or graph coloring solutions) do not balance the color distribution. If there is an insufficient number of elements colored with one or more colors, then an insufficient number of active blocks may be launched resulting in an inefficient use of the GPU resources. Additionally, we expect that the number of colors will not significantly affect the total running time of the algorithm as long as each color owns sufficiently many elements. No matter how the mesh is colored, each element is still only computed once and immediately assembled. This is confirmed in Figure 3 where the performance gain versus the number of colors is shown for a mesh with 4.8M elements. Optimizing the coloring of the mesh is not a primary concern and almost any heuristic should suffice.

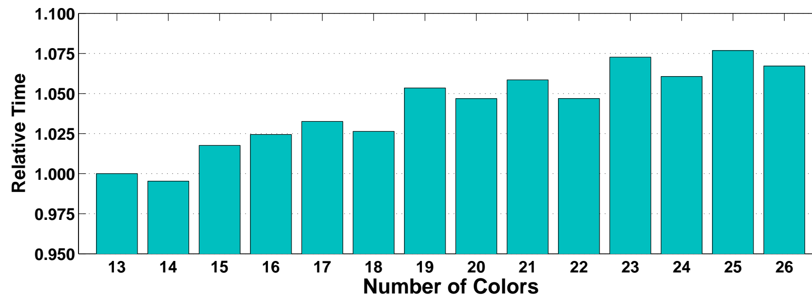


Figure 3. The relative running times obtained from varying the number of colors for a 2D mesh of 4.8M triangular elements on an 8800 GTX GPU. Note that doubling the number of colors results in less than a 10% increase in running time. Minimizing the number of colors is not a primary concern.

4.1.2. Computing and Assembling Colored Elements. Let $\mathcal{E}_k$ be the set of elements colored with color k. Since there are no shared nodes between elements in any one $\mathcal{E}_k$, there is no need to optimize the way the nodal data for each element is retrieved from global memory, as we do in the following sections. Instead, each element simply retrieves its own nodal data from global memory, runs the element kernel to compute the element data, and assembles this data into the system of equations.

To do so, we simply precompute the colored element matrices E_k,

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

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using nmeauth.cls

where $\sigma_k : \mathcal{E}_k \to \{1, \ldots, |\mathcal{E}_k|\}$ is the mapping from global element number to colored element number. These matrices are laid out in memory and threads are assigned to elements so that memory reads can be coalesced. This leads to the parallel algorithm:

1 $tid \leftarrow globalThreadID$; `/* The global thread ID number is the index of the colored element in` E_k `being computed */`
2 **for** *all* $n \in [1, e_n]$ **do**
3     $nodes[n] \leftarrow C(E_k(tid, n))$;
4 Accumulate $(\boldsymbol{A}^{\mathrm{e}}, \boldsymbol{F}^{\mathrm{e}}) = elem(nodes)$ into the system of equations;

This algorithm is run for each color k.

4.1.3. Code and Remarks. How the data is assembled into the system of equations depends on the storage type of the sparse matrix. In this paper we use a general approach and precompute the target index into the system of equations for each entry of the local element data. Since this index map is needed for each element, a simple strategy is to append it to the array of nodal indices associated to each element. To have coalesced reads, the information is arranged as shown in Figure 4.

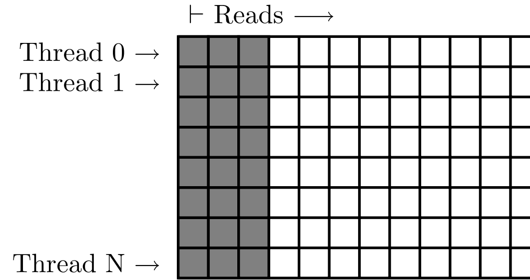


Figure 4. The column-major matrix storing the node numbers (gray) of each element is followed by indices into the system of equations, mapping entries of the element data into the system of equations. For nodes with one degree of freedom and symmetric element data, 9 (6 from $\boldsymbol{A}^{\mathrm{e}}$ and 3 from $\boldsymbol{F}^{\mathrm{e}}$) indices are needed.

When N is a multiple of 32, this matrix is read with fully coalesced reads from global memory. In the case that there are fewer than N elements to be computed by this block, we pad with (-1)s to preserve coalescing by retaining memory alignment. The CUDA code is then given below.

```
__global__ void assemblyElemColor(float* AF, NodalData* C, int* Ek)
{
    // AF stores the matrix and right hand side data

    // Compute this thread's pointer into Ek
    Ek += blockIdx.x * blockDim.x * (NODESPERELEM + ELEMDATASIZE) + threadIdx.x;

    int nPtr = Ek[0];
```

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using nmeauth.cls

```
    // Check that there is some work to do
    if( nPtr != -1 ) {
        // Do one element per thread (could extend to more than one)
        NodalData nodes[NODESPERELEM];

        // At this point, Ek holds the nodal data (gray squares in figure 4).
        nodes[0] = C[nPtr];
        Ek += blockDim.x;

#pragma unroll
        for( int k = 1; k < NODESPERELEM; ++k ) {
            nodes[k] = C[ Ek[0] ];
            Ek += blockDim.x;
        }

        // Ek[0*blockDim.x] ... Ek[(ELEMDATASIZE-1)*blockDim.x]
        // hold indices mapping the element data into AF.
        // This corresponds to the white squares in Figure 4.
        // Compute and assemble at the same time:
        elem( nodes, AF, Ek );
    }
}
```

If the storage of the sparse matrix is known and a simpler indexing scheme that uses less memory and fewer transactions with global memory can be used then significant speedups may be possible.

We expect the Assembly-by-Element approach to be relatively slow since there is no sharing of information between threads. A possible optimization consists in writing the element data to shared memory in an attempt to make coalesced writes to the system of equations and save a small amount of memory transactions, but this does not appear to be wholly economical. In the following section, we discuss the assembly by non-zero entries using global memory, which is similar to the coloring algorithm, but with some modifications that allow it to outperform the latter when the input nodal data requirement of the element subroutine or output element data become large with respect to the available shared memory.

4.2. Assembly by Non-Zero Entries Using Global Memory

One possible mapping of the direct stiffness method to the NVIDIA CUDA API would assign one thread to compute the element data for one mesh element and, to avoid race conditions, write the element data to global memory for later reduction into the system of equations. Since the reduction would then operate on element data stored in global memory and since there are no global synchronization primitives, the assembly must be performed using a separate kernel.

Thus, we discuss the element calculation kernel and the assembly kernel separately. In section 4.2.1 we introduce a very simple element calculation kernel which we optimize in section 4.2.2. Section 4.2.3 then discusses one possible approach to the reduction kernel.

4.2.1. Simple Element Calculation Kernel. A straightforward version of the calculation kernel is given in Algorithm 2, in which we assign one thread to one element, gather the nodal data, compute the element data, and write it to global memory. We can coalesce the reads from the element matrix E via an appropriate precomputed reshuffling of the entries, and coalesce the writes of the element data to global memory.

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using nmeauth.cls

```
1 tid ← globalThreadID;
2 for all unrolled n ∈ [1, e_n] do
3   nodes[n] ← C(E(tid, n));
4 gMem ← elem(nodes);                    /* Store in global memory */
```

Algorithm 2: Simple element calculation kernel using global memory.

In general, the reads from the nodal data matrix are not coalesced. It is possible to define an "element nodal data matrix" C^{e} where $C^{\mathrm{e}}(e,a)$ is the nodal data of the local node a of element e, eliminating the need to read from E and providing the nodal data in coalesced reads. If the nodal data does not change, this may work and may lead to a speed-up, depending on the specific problem being solved. If the nodal data changes, this matrix needs to be updated at each step. Compared to the approach described below, this update will require an extra pass through global memory (reading and writing C^{e}), and therefore will be, in general, slower.

4.2.2. Optimizing the Element Calculation Kernel. A significant optimization can be 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 elements to be computed.

If $\mathcal{E}_k$ is the set of elements that the kth thread block is responsible for computing, then we compute the set of nodes $\mathcal{N}_k$,

$$\mathcal{N}_k = \{E(e,a) : e \in \mathcal{E}_k,\ a = 1, \ldots, e_n\},$$

that thread block k will need to retrieve for its element subroutines. Note that $\mathcal{N}_k$ does not form a partition of the set of nodes but $\mathcal{E}_k$ does form a partition of the elements. These nodes are to be fetched from global memory and stored in shared memory to be used by the elements of $\mathcal{E}_k$. The finite size of the shared memory space imposes a strict limit on the cardinality of $\mathcal{N}_k$. This limit is

$$\forall k, \quad |\mathcal{N}_k| \le \frac{|S_{\mathrm{f}}|}{|D_{\mathrm{n}}|} \tag{8}$$

where $|S_{\mathrm{f}}|$ is the size of the shared memory and $|D_{\mathrm{n}}|$ is the size of the data for a single node, both measured in number of single-precision floating-point numbers. Typically, for nonlinear problems, $|D_{\mathrm{n}}| = n_d + n_f$ because each node contributes n_d coordinates and the n_f field values from the last approximation to the solution in a Newton-Raphson iteration. In practice, for nodes in 3D with one degree of freedom per node, over 800 nodes can be stored in the 16K shared memory space. By adopting a finite element space of continuous functions that are affine in each tetrahedral element, the size of the element set $\mathcal{E}_k$ is over 3000.

We therefore need to determine a partition of the elements in the mesh in groups of roughly equal number of elements such that constraint (8) is satisfied. This last condition is more likely to be satisfied if we request the partition of elements to minimize $\max_k |\mathcal{N}_k|$. Good partitions under these conditions can be obtained from conventional graph partitioning software such as `METIS` [18]. In our case, we partitioned the connectivity graph of the mesh, or dual graph to the mesh, in which each element is a node of the graph and two nodes are connected by an edge if the corresponding elements of the primal mesh share a face. The resulting partitions have roughly the same number of elements each, and minimize the total number of element

Copyright © 2000 John Wiley & Sons, Ltd.

neighbors across partitions. While this is not exactly a minimization of $\max_k |\mathcal{N}_k|$, it generally is an acceptable solution. Given the strict limit on $|\mathcal{N}_k|$, accurate estimates of the number of element partitions to request from METIS can be made, but multiple partition attempts may be required to maximize $|\mathcal{E}_k|$ while satisfying constraint (8).

With $\mathcal{N}_k$ defined, the nodal data can be gathered into shared memory. We now need to know which nodes to pass to the element subroutine. For each thread block k, we precompute block element matrices, E_k^ℓ, defined by

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

where $\sigma_k : \mathcal{N}_k \to \{1,\dots,|\mathcal{N}_k|\}$ is the mapping of a global node number $E(e,a)$ to its block node number $E_k(e,a)$ within block k, which can be used to find the nodal data in shared memory. Then, for each block k, we arrange the e_n-tuples into lists which will be read fully coalesced from global memory. The result is a column-major matrix with $blockSize$ rows and data profile shown in Figure 5 for the case $e_n = 3$. A thread will read e_n integers in coalesced memory transactions, and pass to the element subroutine the 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.

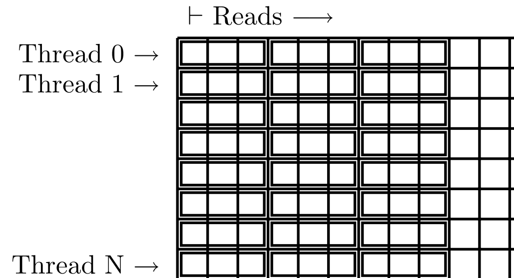


Figure 5. The column-major block element matrix E_k in the case $e_n = 3$. Each entry stores a block node number $E_k^\ell(e,a)$ which can be used to find the nodal data in shared memory. Each group of 3 entries defines an element to be computed. A thread block will read down a column of the array in coalesced memory transactions.

The resulting algorithm for the first stage of the global assembly method is sketched in Algorithm 3. Therein, $end_k = blockSize\ \lceil |\mathcal{E}_k|/blockSize \rceil$ ($\lceil\,\rceil$ is the ceil function), $sMem$ denotes the shared memory space and $gMem$ denotes the global memory space. Note that all reads from the block local element matrix are coalesced. The element subroutine is responsible for writing to global memory with coalesced transactions.

4.2.3. Assembly and Reduction from the Element Data in Global Memory. One advantage of the global assembly methods is that the computation of the element data and the accumulation of the element data into the system of equations are decoupled and will be performed in two separate kernels.

In this paper, we have implemented a simple accumulation technique. We determine the indices into the element data array previously stored in $gMem$ that contribute to each non-zero entry of the system of equations. Each of these lists is appended with the index into the system of equations of the NZ in question. Thus, we have NZ reduction lists that look like

Copyright © 2000 John Wiley & Sons, Ltd.

```
1 k ← blockID;
2 Fetch all nodal information in C for nodes in N_k, and store in sMem;
3 —— Barrier ——                      /* All threads in the block synchronize */
4 tid ← blockThreadID;
5 while tid < end_k do
6     for all n ∈ [1, e_n] do
7         nodes[n] ← sMem[E_k[tid]];
8         tid += blockSize;
9     gMem ← elem(nodes);
```

Algorithm 3: Global assembly with shared prefetching.

where the light entries represent indices into the element data array (source indices) and the final dark entry is the index of the NZ of the system of equations (target index). These two interpretations of the integer being read can be distinguished by any kind of flag. We chose to negate the target index and subtract 1 (inverting all bits in the integer under a two's complement system). Additionally, we add 1 to all source indices. Thus, the sign of the integer distinguishes the two types of indices. When a thread reads a list of this form, it will accumulate all values pointed to in the source array S by the source indices and store the result in the target array T at the target index (see Algorithm 4).

```
1 for all i in NZ_Reduction_List do
2     if i > 0 then
3         t ← t + S[i − 1];
4     else if i < 0 then
5         T[−i − 1] ← t;
```

Algorithm 4: Serial reduction operation from reduction array.

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 work loads. Furthermore, we need to coalesce the access to these lists, which requires interweaving the lists and some padding of the shorter lists with zeros (null operations in the previous algorithm). Consequently, such a strategy leads to significant memory and possibly entire memory transactions wasted. 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 and interweave the NZ reductions lists into an array that will be read fully coalesced in the kernel. Simple packing algorithms such as Largest-Processing-Time (LPT) [19] appear to suffice and result in small amounts of wasted space. The result is a column-major matrix with *blockSize* rows and data profile shown in Figure 6. The parallel algorithm is given in Algorithm 5.

For efficiency, we assemble contiguous NZs of the system of equations in shared memory, then perform a coalesced write into the system of equations. Thus, the only memory reads

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using nmeauth.cls

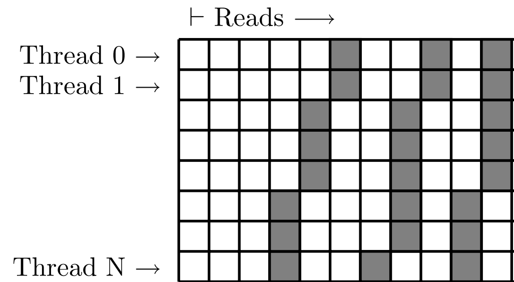


Figure 6. The column-major reduction matrix with *blockSize* rows. The light entries represent source indices and the dark entries represent target indices. A thread block will read down a column of the array in coalesced memory transactions.

```
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;
```

Algorithm 5: 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. This same approach will be used in section 4.3.2 with an alternate source array.

that are not coalesced are the retrievals of the element data.

4.2.4. Summary and Remarks. The entire assembly algorithm by NZ using global memory is diagrammed in Figure 7.

This version of the algorithm can be further optimized for higher order elements by using the shared memory only for nodes that are shared between two or more elements. That is, the interior nodes of an element should be read on the fly from global memory, conserving the shared memory and maximizing the cardinality of $\mathcal{E}_k$ by reducing the size of $\mathcal{N}_k$.

In the case that the global memory cannot hold all the element data, this algorithm can easily be split into multiple passes. This is analogous to suggestions in [11]. Each pass would compute the element data for some subset of elements and accumulate them into the system of equations in precisely the same way.

More advanced encodings of the block element matrix can be performed when strict limits on the magnitude of entries are known. For example, if all entries of the block element matrix

Copyright © 2000 John Wiley & Sons, Ltd.

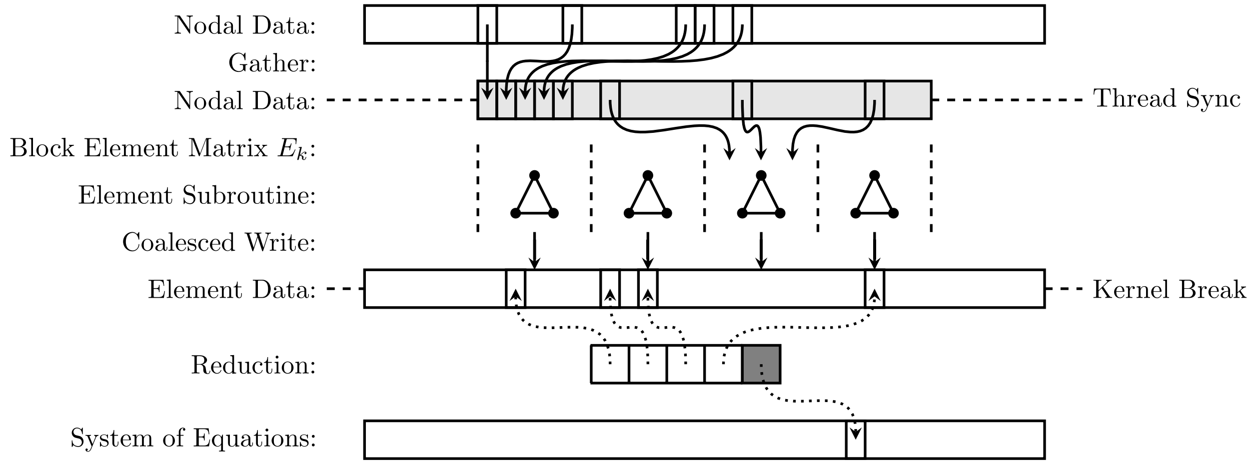


Figure 7. The global assembly by NZ on the GPU. Global memory is depicted in white and shared memory is depicted in light gray. Solid black arrows represent memory reads and writes and dotted arrows represent references to memory. The kernel break denotes the portion end of the element computation kernel and the beginning of the assembly kernel.

with $e_n = 3$ are known to be less than $1024 = 2^{10}$, then three entries can be encoded into a single 32-bit integer and retrieved in one memory transaction.

Finally, the assembly of the element data can be expressed as the sparse-matrix vector multiply

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

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. 6. Thus, this stage of the global assembly method should take advantage of the emerging research in sparse matrix-vector multiplication methods (SpMV) on GPUs. Some recent research into SpMV is discussed in section 5.2 with a number of provided references.

4.3. Assembly by Non-Zero Entries Using Shared Memory

Using the shared memory to store element data has advantages from both the local memory assembly methods and global memory assembly methods. Like local assembly methods, the element data is stored in a memory space that has very fast reads and writes compared to the global memory. Like global assembly methods, we can share element data across threads in order to reduce the total number of calls to the element subroutine. In this section, we detail the most successful of our assembly methods for low-order finite element problems on unstructured meshes.

First, because there can be no interblock communication between threads, we partition the nodes so that each block of threads is responsible for a set of nodes of the mesh (or rows of the system of equations). If $\mathcal{N}$ is the set of all nodes, then for a given partition $\mathcal{N}_k \subset \mathcal{N}$ we must find the set of elements

$$\mathcal{E}_k = \{e \in \mathcal{E} \mid \exists\, a \text{ such that } E(e, a) \in \mathcal{N}_k\},$$

Copyright © 2000 John Wiley & Sons, Ltd.

that is, the set of elements adjacent to any node of $\mathcal{N}_k$. Note that $\mathcal{E}_k$ does not form a partition of the set of elements. Instead elements of $\mathcal{E}_k$ whose adjacent elements are not all members of $\mathcal{E}_k$ are most likely contained in at least one other $\mathcal{E}_{k'}$, $k' \neq k$.

Using the shared memory space to store the element data imposes a strict limit on the cardinality of $\mathcal{E}_k$, which complicates the partitioning step. This limit is

$$|\mathcal{E}_k| \leq \frac{|S_{\mathrm{f}}|}{\max(|D_{\mathrm{n}}|\,, |D_{\mathrm{e}}|)} \qquad \forall k \tag{9}$$

where $|S_{\mathrm{f}}|$ is the size of the shared memory, $|D_{\mathrm{n}}|$ is the size of the input nodal data, and $|D_{\mathrm{e}}|$ is the size of the element data $\boldsymbol{A}^{\mathrm{e}}$ and $\boldsymbol{F}^{\mathrm{e}}$ from an element subroutine, all measured in single-precision floating-point numbers. In a nonlinear problem where each node of an element contributes n_d coordinates and n_f fields we have $|D_{\mathrm{n}}| = e_n(n_d + n_f)$. Additionally, $|D_{\mathrm{e}}| = e_f(e_f + 1)$, which is the combined size of the local element matrix $\boldsymbol{A}^{\mathrm{e}}$ and force vector $\boldsymbol{F}^{\mathrm{e}}$. If the element matrix is known to be symmetric, then $|D_{\mathrm{e}}| = \frac{1}{2}e_f(e_f + 3)$. These values are tabulated for the 2D triangular element and 3D tetrahedral element of polynomial order N and $n_f = 1$ in Table IV.

	2D triangles				3D tetrahedrons			
N	e_n	$e_n(n_d+n_f)$	$e_f(e_f+1)$	$\frac{1}{2}e_f(e_f+3)$	e_n	$e_n(n_d+n_f)$	$e_f(e_f+1)$	$\frac{1}{2}e_f(e_f+3)$
1	3	9	12	9	4	16	20	14
2	6	18	42	27	10	40	110	65
3	10	30	110	65	20	80	420	230
4	15	45	240	135	35	140	1260	665
5	21	63	462	252	56	224	3192	1652
6	28	84	812	434				

Table IV. Storage requirements for elements with $n_f = 1$ and polynomial order N.

Thus, in order to minimize the total number of element subroutine calls, we want the sets $\mathcal{E}_k$ to minimize $\sum_k |\mathcal{E}_k|$. Good partitions of the set of nodes under these conditions can be obtained from conventional graph partitioning software such as `METIS` [18]. In this case we partition the graph formed by the nodes and the edges of the mesh.

4.3.1. Prefetching and the Scatter Array. Because the elements of $\mathcal{E}_k$ share many nodes, an important optimization is to retrieve the required nodal data only once from global memory. Although we previously described an efficient gather operation of this sort in section 4.2 which can be modified for this case, we present an alternative method here. The nodal data can be retrieved once and distributed to each element that requires it. This allows us to use the same shared memory space to store the nodal data and the resulting element data. Since the limiting factor is the size of the output element data rather than the input nodal data (see Table IV) this does not change the partitioning strategy discussed above.

Given a set of elements, we need to retrieve the needed nodal data and distribute that data to the element subroutines. First, we compute the set of required nodes for each block. For the set of elements $\mathcal{E}_k$ that the kth thread block is responsible for computing, we compute the corresponding set of nodes $\overline{\mathcal{N}}_k$,

$$\overline{\mathcal{N}}_k = \{E(e,a) : e \in \mathcal{E}_k,\ a = 1, \ldots, e_n\},$$

Copyright © 2000 John Wiley & Sons, Ltd.

that thread block k needs to retrieve. Clearly, $\mathcal{N}_k \subset \overline{\mathcal{N}}_k$. (Note that $\overline{\mathcal{N}}_k$ does not form a partition of the nodes.) Figure 8 shows an example of the sets $\mathcal{N}_k$, $\overline{\mathcal{N}}_k$ and $\mathcal{E}_k$.

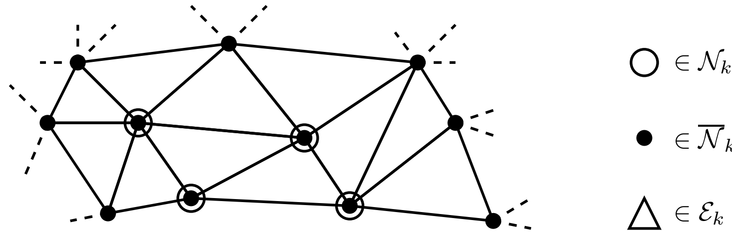
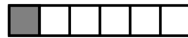


Figure 8. Example of the sets $\overline{\mathcal{N}}_k$, $\mathcal{N}_k$, and $\mathcal{E}_k$. As $|\mathcal{E}_k|$ increases, the ratio $|\overline{\mathcal{N}}_k| / |\mathcal{N}_k|$ decreases, which improves the efficiency of the algorithm.

Second, the nodes of $\overline{\mathcal{N}}_k$ are to be fetched from global memory and stored in shared memory. 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. In the spirit of the reduction lists in section 4.2.3, for each node we create a list of the form



where the dark entry represents the node number (source index) and the light entries represent the block element number that requires the data from that node (target index). Again, the two types of integers can be distinguished by any kind of flag, and we have chosen to use negation minus one for the target indices and adding one to the source indices. When a thread reads a list of this form, it will fetch from the source array S the value pointed to by the source index and store this value in the target array T at all of the target indices (see lines 6 to 9 of Algorithm 6).

The lists associated to different nodes in $\overline{\mathcal{N}}_k$ may have significantly different lengths. As in section 4.2.3, assigning one thread to one list could lead to unbalanced work loads, wasted memory and wasted memory transactions. Instead, we pack these lists into a *scatter array* using a packing algorithm such as LPT. The result is a column-major matrix with *blockSize* rows and data profile shown in Figure 9.

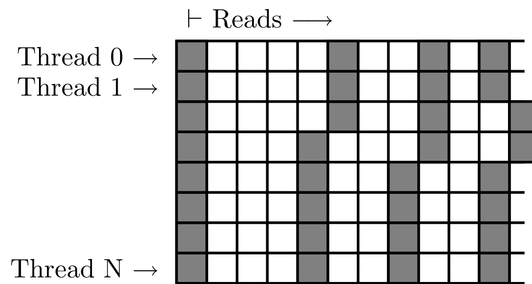


Figure 9. The column-major scatter matrix with *blockSize* rows. The dark entries represent source indices and the light entries represent target indices. A thread block will read down a column of the array in coalesced memory transactions.

Copyright © 2000 John Wiley & Sons, Ltd.

The parallel algorithm to perform the instructions in the scatter array is then particularly simple and is shown in Algorithm 6.

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

Algorithm 6: Gather-scatter operation from scatter array. In this case, the source array S is the nodal data array in global memory and the target array T is the shared memory space.

It should be noted that end_k should be either read from constant memory so that a broadcast transaction is used or retrieved from global memory only once by a particular thread which broadcasts it to the other threads in the block via a small shared memory scratch space.

4.3.2. Assembly and the Reduction Array. After the element subroutines have calculated the element data and stored it into shared memory, overwriting the nodal data, we need to take the element data and assemble it 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. Therefore, we can use the same reduction plan as described in section 4.2.3, but with the source array now being the shared memory space rather than the global memory space.

4.3.3. Summary and Remarks. The entire procedure is diagrammed in Figure 10. The shared assembly by NZ algorithm is heavily constrained by the size of the shared memory space. For low order elements, for which the required input nodal data and output element data size is small, many elements will fit into the shared space. This allows the partition size to be large and results in a more efficient algorithm since fewer elements must be computed redundantly between blocks. However, when the element data size increases – as for higher order elements – few elements fit into the shared memory space and it becomes difficult or impossible to find a good partition of the degrees of freedom.

A optimization that could alleviate this limitation would be to perform the assembly by using multiple passes to compute all the elements of $\mathcal{E}_k$. That is, compute some elements of $\mathcal{E}_k$ in shared memory, assemble the element data, and continue to other elements of $\mathcal{E}_k$. There are multiple, complicated choices that must be made in an implementation that uses this strategy. It will not be addressed here.

To avoid shared memory bank conflicts, the shared memory assigned to each element does not have to be contiguous and can instead be interweaved.

We note the versatility of using this scatter and reduction array approach. It can easily be modified to any sparse matrix structure that is desired to be used for the solution stage. If $\boldsymbol{A}$

Copyright © 2000 John Wiley & Sons, Ltd.

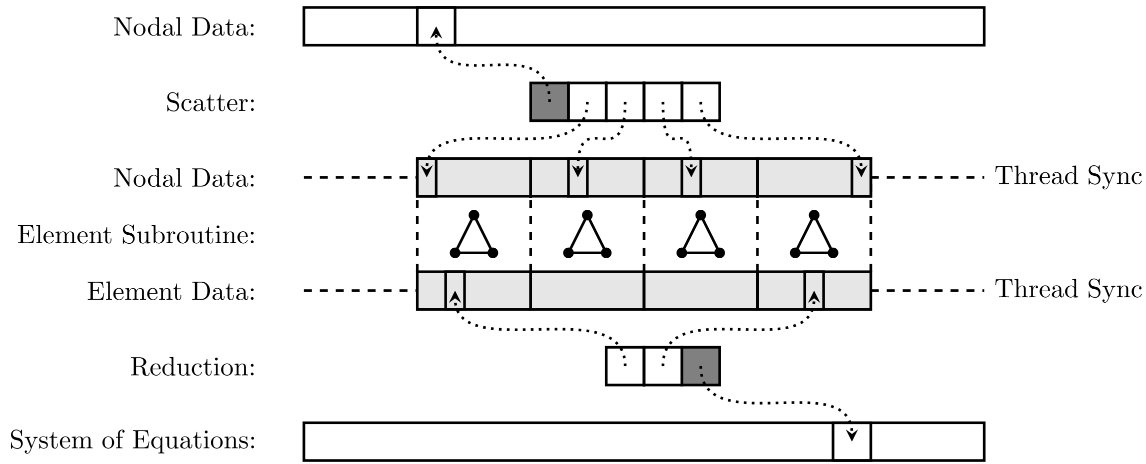


Figure 10. The shared assembly by NZ on the GPU. Global memory is depicted in white and shared memory is depicted in light gray. Dotted arrows represent references into memory.

is known to be symmetric, we can either fill only the upper or lower half of $\boldsymbol{A}$ or fill the entire matrix with a modification to the reduction array instructing the assembly kernel to save the reduced result in more than one target index.

4.4. Assembly by Non-zero Entries Using Local Memory

Finally, we considered an approach in which each thread uses its registers and the memory local to it to take sole responsibility for assembling one NZ of the system of equations at a time, as mentioned in section 3.4. Each thread calls the element subroutine with appropriate input nodal data as needed by the NZ. This approach was optimized with techniques similar to those in the previous sections: mesh partition and prefetching of shared nodal data and packing NZ dependency lists into a coalesced matrix. Although the amount of redundant data and computation prevents competitive performance with the other methods, it is included in the following analysis for completeness.

Although this approach requires significant redundant computation when the element kernel is treated as a black box, it may be an appropriate choice when the definition of $\boldsymbol{A}^{\mathrm{e}}$ and $\boldsymbol{F}^{\mathrm{e}}$ can be simplified, as was the case in [9].

4.5. Discussion of Assembly Methods

Each assembly method presented in this paper has advantages and limitations. The appropriate method in a given situation depends on a number of considerations including the mesh size and the element's memory and computational requirements. Some of these main concerns are tabulated in Table V.

Copyright © 2000 John Wiley & Sons, Ltd.

Assembly Method	Limiting Resource	Advantages	Limitations
ElemColor (4.1)	Global memory bandwidth	Common approach. Minimum element subroutine calls.	Little utilization of hardware specific optimizations.
GlobalNZ (4.2)	Global memory space	Minimum element subroutine calls. Element computation and assembly disjoint.	Global memory prevents scaling to very large number of elements.
SharedNZ (4.3)	Shared memory space	Most utilization of hardware specific optimizations.	Small shared memory prevents scaling to large output element data.
LocalNZ (4.4)	Peak floating point operations per second (flops)	Can be competitive when certain problem specific optimizations apply.	Computational redundancy prevents scaling to elements with large computational requirements.

Table V. Table of advantages and limitations of each method. The column “Limiting Resource” lists the hardware resource that is typically strained the most for that method.

Copyright © 2000 John Wiley & Sons, Ltd.
Prepared using **nmeauth.cls**

We note that only the NZ assembly procedures were considered. Although we can show that small gains are available in certain cases by using an assembly by row approach, we find that formulating these methods in terms of NZs is more clear and computationally versatile. Regardless, the methods described in this paper should be considered as a starting point for the repeated assembly of systems of equations stemming from FEM models on the GPU. More advanced and problem-specific optimizations should be made as needed.

5. Solving the Sparse Linear System

After assembling the system of equations $\boldsymbol{A}\boldsymbol{u} = \boldsymbol{F}$, we need to solve it for the unknowns $\boldsymbol{u}$. Of course, the system can constructed explicitly, passed back to the host CPU, and solved using any of the many optimized solvers widely available. Alternatively, unstructured sparse matrix solvers on the GPU have been under intense research recently and have achieved significant speedups over their CPU counterparts; see Section 5.2 below. Direct or iterative solvers can be used on the explicitly constructed matrix equations. Finally, the above algorithms can be applied in so-called matrix-free iterative methods in which the matrix equations are never explicitly constructed but rather the assembly algorithms are used to perform sparse matrix vector products.

5.1. Direct Solvers

Although direct solvers such as Cholesky decomposition, Gauss-Jordan Elimination, QR decomposition, and LU decomposition have been shown to be successful for dense matrices on the GPU and the NVIDIA CUDA API specifically [20, 15], the computing architecture does not appear ideally suited for direct solvers of general sparse matrices. Due to the fill-in, reordering, and need for dynamically updatable sparse matrix structures that many direct solvers require, efficient parallel implementations are difficult and, to our knowledge, no efficient GPU implementation is available at this time.

5.2. Iterative Solvers

Significant research has been performed in adapting conjugate gradient and multigrid techniques to the GPU [8, 21, 7]. The bottleneck in the conjugate gradient method is the sparse matrix-vector multiplication (SpMV), which several papers have recently addressed. Buatois et al. [22] present a general sparse linear solver deemed the Concurrent Number Cruncher (CNC) which implements a preconditioned conjugate gradient solver using block compressed row storage for improved register blocking, but can result in non-optimal global memory accesses. Bell et al. [23] compare the performance of the SpMV under many sparse storage formats and sparsity patterns, from regular to highly irregular. Baskaran et al. [24] provide a highly optimized SpMV kernel specifically for the CUDA API which uses a storage structure that improves data access patterns and reuse. Results from [25] may be incorporated into some of these approaches to further accelerate gather and scatter operations.

Copyright © 2000 John Wiley & Sons, Ltd.

6. Experimental Results

6.1. Numerical Validation

To validate the FEM assembly procedures implemented on the GPU, we chose to construct the system of equations for a simple steady heat equation

$$\begin{aligned} \nabla \cdot (\boldsymbol{\kappa} \cdot \nabla u(\boldsymbol{x})) &= f(\boldsymbol{x}) \quad \boldsymbol{x} \in \Omega \\ u(\boldsymbol{x}) &= g(\boldsymbol{x}) \quad \boldsymbol{x} \in \Gamma_g \\ \boldsymbol{n}(\boldsymbol{x}) \cdot \boldsymbol{\kappa} \cdot \nabla u(\boldsymbol{x}) &= q(\boldsymbol{x}) \quad \boldsymbol{x} \in \Gamma_q \end{aligned}$$

on a the domain pictured in Figure 11 with $\Gamma_g = \Gamma_{g_1} \cup \Gamma_{g_2}$, $g(\boldsymbol{x}) = 200$ on Γ_{g_1}, $g(\boldsymbol{x}) = 10$ on Γ_{g_2}, $f(\boldsymbol{x}) = 0.1$, and $q(\boldsymbol{x}) = 0$. Here $\boldsymbol{\kappa}$ is the thermal conductivity matrix, which we take as the identity in our examples. The domain is meshed with various levels of refinement. The largest mesh has 2.4 million nodes and 4.8 million elements.

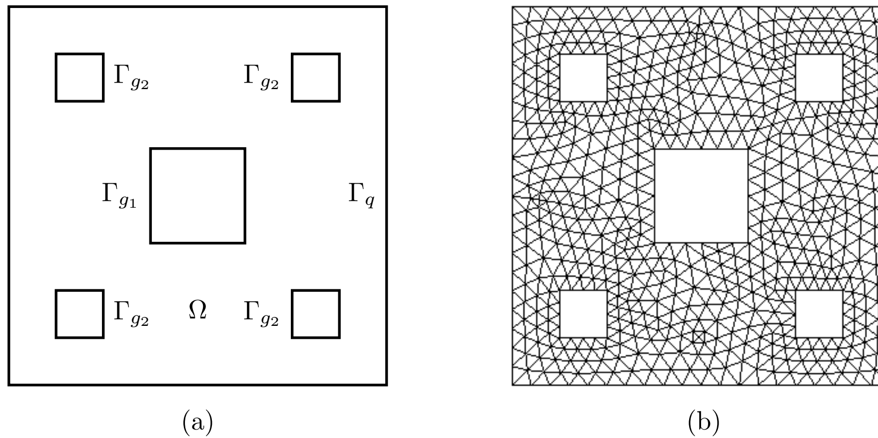


Figure 11. The domain of the finite element test problem and an example mesh.

Due to the current necessity to use single-precision arithmetic on the GPU, assembling the system of equations leads to different results if performed with double precision in the host CPU, or with single precision in the GPU. Table VI lists the average relative error between the entries of the system of equations constructed in double-precision on the host and single precision on the device. Errors are similar between the GPU assembly methods and increase with the inverse of the characteristic mesh size of the grid, $h \sim |\mathcal{N}|^{-1/2}$. 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 $1/h$. It should be noted that for d-dimensional finite element models, the characteristic mesh size scales like $h \sim |\mathcal{N}|^{-1/d}$. Thus, for higher dimensional finite element problems, the accuracy of the system of equations computed using single-precision arithmetic increases for a given $|\mathcal{N}|$.

Table VII lists the relative error between the solutions of the system of equations from the double-precision host and the system of equations from the single-precision device. Both

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using nmeauth.cls

Mesh (Nodes,Elements)	GPU Assembly Method: ElemColor	SharedNZ	LocalNZ	GlobalNZ
184,292	1.6e-7	1.6e-7	1.6e-7	1.6e-7
664,1168	3.0e-7	3.0e-7	3.0e-7	3.0e-7
2.5K,4.7K	6.2e-7	6.2e-7	6.2e-7	6.2e-7
9.7K,18.7K	1.2e-6	1.2e-6	1.2e-6	1.2e-6
38K,75K	2.6e-6	2.6e-6	2.6e-6	2.6e-6
150K,299K	5.2e-6	5.2e-6	5.2e-6	5.2e-6
600K,1.2M	1.1e-5	1.1e-5	1.1e-5	1.1e-5
2.4M,4.8M	2.1e-5	2.1e-5	2.1e-5	2.1e-5

Table VI. The average relative error of the entries in the system of equations, $(\sum_{ij} |A^s_{ij} - A^d_{ij}|/|A^d_{ij}| + \sum_i |F^s_i - F^d_i|/|F^d_i|)/(NZ + N)$, between the single precision system computed on the device, $\boldsymbol{A}^s, \boldsymbol{F}^s$, and the double precision system computed on the host, $\boldsymbol{A}^d, \boldsymbol{F}^d$.

are solved using a conjugate gradient solver on the host in double-precision to approximately machine precision. Thus, this represents the true difference between the single-precision finite element model and the double-precision finite element model of the original problem and is used as a measure of the appropriateness of using single-precision in this case.

Mesh (Nodes)	184	664	2.5K	9.7K	38K	150K	600K	2.4M
ε	1.4e-7	1.7e-7	5.0e-7	8.2e-7	1.3e-6	3.7e-6	8.2e-6	1.2e-5

Table VII. The relative error $\varepsilon = \left\|\boldsymbol{u}^s - \boldsymbol{u}^d\right\|_2 / \left\|\boldsymbol{u}^d\right\|_2$ of the solution of the single-precision system of equations $\boldsymbol{A}^s\boldsymbol{u}^s = \boldsymbol{F}^s$ from the device and the double-precision system of equations $\boldsymbol{A}^d\boldsymbol{u}^d = \boldsymbol{F}^d$ for solutions $\boldsymbol{u}^s$ and $\boldsymbol{u}^d$ of precision $\|\boldsymbol{F}^s - \boldsymbol{A}^s\boldsymbol{u}^s\|_2^2 < 10^{-12}$ and $\left\|\boldsymbol{F}^d - \boldsymbol{A}^d\boldsymbol{u}^d\right\|_2^2 < 10^{-12}$.

6.2. Element Kernels

For our test case, we write element kernels to compute

$$A^{\mathrm{e}}_{ij} = \int_{\Omega^{\mathrm{e}}} \boldsymbol{\kappa}\nabla\varphi_i \cdot \nabla\varphi_j \; d\Omega \qquad\qquad F^{\mathrm{e}}_i = \int_{\Omega^{\mathrm{e}}} \varphi_i f \; d\Omega - \sum_j A^{\mathrm{e}}_{ij} g_j$$

where

$$g_j = \begin{cases} 0 & \text{if } \boldsymbol{x}_j \in \Omega \\ g(\boldsymbol{x}_j) & \text{if } \boldsymbol{x}_j \in \Gamma_g \end{cases}$$

is a vector used to impose essential boundary conditions, as explained in Section 3.3. Other options are possible.

For this paper, we used N^{th} order Lagrange triangular elements (isoparametric, but all triangles have straight edges). The element subroutine was algebraically optimized to minimize memory usage and take nearly maximal advantage of the GPU's ability to perform multiply-adds in a single clock cycle. Constant values, such as parent domain basis function values and derivatives, are not read from memory, but instead hard-coded into the source code.

Copyright © 2000 John Wiley & Sons, Ltd.

The element kernels were written in a modular fashion and are therefore almost identical for all methods. There were however minor differences. For example, the SharedNZ and GlobalNZ algorithm both know where the element data is to be stored simply from the element number whereas the ElemColor algorithm must look up an index into global memory to accumulate each element datum into the system of equations as it is computed.

Extensions to more general classes of elements, such as isoparametric elements, will involve a more complicated element subroutine that may require additional memory. The Jacobian matrix may not be constant over the element Ω^e, and may have to be recomputed at each quadrature point. It remains important to minimize the memory usage of the element kernel. As we will see in section 6.3, computations on unstructured meshes are heavily memory bound and the use of off-chip local memory in the element kernel compounds the issue. Instead, element kernels should be written to minimize the memory usage, even at the cost of additional computation.

6.3. Performance Analysis

Our experimental setup is composed of an NVIDIA GeForce 8800 GTX processor and an NVIDIA Tesla C1060 processor installed on the PCI Express 2 bus (8 GB/s bandwidth) of a Intel Core 2 Quad CPU Q9450 2.66GHz with 8GB of RAM and running Linux kernel 2.6.28. The 8800 GTX card has 16 multiprocessors, 128 cores, 768MB of memory, with a memory bandwidth of 86.4GB per second. The C1060 card has 30 multiprocessors, 240 cores, 4GB of memory, with a memory bandwidth of 102GB per second. We use CUDA version 2.3, driver 190.18, `gcc` version 4.3.3, and `nvcc` release 2.3 version 0.2.1221.

The reference code written for the host followed precisely algorithm 1. Optimizations were made to prevent superfluous memory accesses to achieve a speedup of approximately 2x. The reference code is compiled with the `-O2` flag.

The running times of a number of the more successful GPU algorithms are shown in Figure 12 and Figure 13. These running times are measured in wall time with appropriate `cudaThreadSynchronize` calls after CUDA kernels to ensure accurate timing. Thus, these times include GPU call overhead, but do not include the precomputation time or any GPU-CPU data transfers. The test case is the 2D heat equation in Section 6.1.

The relative running times of the algorithms are presented as a function of the mesh size in Figure 12. From the figure, we note that when the mesh is very small, the algorithms do not run with sufficient occupancy on the device. When the mesh partition size is small, the thread blocks are unable to share large amounts of data. Furthermore, at the smallest mesh sizes, the CUDA overhead of calling the assembly kernels on the device become significant – about 30% of the recorded time for the SharedNZ algorithm applied to the smallest mesh, determined empirically. As the mesh size increases, the computational occupancy of the device increases, and the compute time per element stabilizes for the device algorithms. The serial assembly run on the host shows an increase in the compute time per element as the mesh size continues to increase due to an increase in cache misses when accumulating data into the system of equations. This could likely be resolved with an appropriate reordering of the nodes to improve data locality on the host. From the figure, it is clear that the GPU only provides speedups when the occupancy of the device is sufficiently high.

Figure 13a shows the relative running times of the finite element assembly procedures on the host and an 8800 GTX GPU (NVIDIA compute capability v1.0) versus the order of the

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using nmeauth.cls

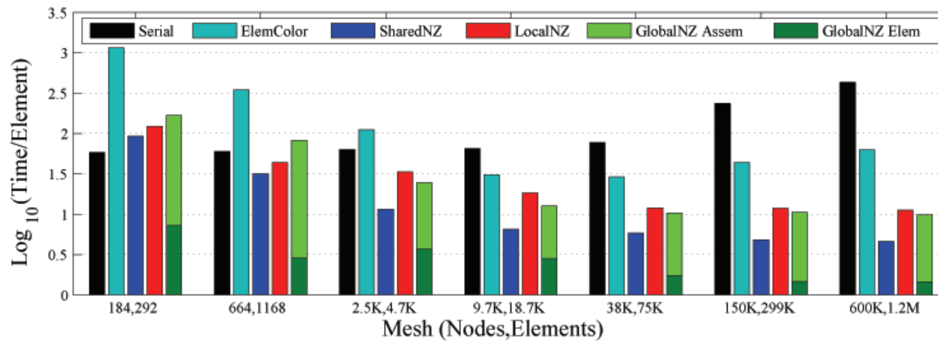


Figure 12. Running time versus size of the mesh.

element being used. The primary implication of using higher order elements is that more data is required on input and output, straining the shared and local memory requirements of each approach on the device. As we can see in the plot, the SharedNZ algorithm achieves a speed up of approximately 35x to the serial version, but only for element order 1 and 2. At element order 3 and 4 GlobalNZ has a comparable speedup, around 10-20x. At element order 5, the SharedNZ algorithm fails due to the inability to partition the mesh into pieces small enough to fit into the shared memory space.

Figure 13b shows the relative running times of the same methods, but with a dummy element kernel that performs no significant calculations. This dummy kernel reads in the same input data and writes to output the result of a very simple arithmetic expression making use of all the input data to prevent the CUDA compiler from suppressing them. The input and output data profiles are exactly the same, only the amount of computation in the element kernel is reduced. This allows us to approximately determine how much of the recorded time is in the actual element kernel computation and how much is due to the memory latencies of the device. As shown in Figure 13c little to no speedup is obtained for all methods at low order. At order 4 the SharedNZ speedup is approximately 1.2 and at order 5 the LocalNZ and GlobalNZ Elem speedup is approximately 1.6. Therefore, the GPU methods are very strongly bandwidth-bound: performance is limited by the memory bandwidth of the card and so cannot be improved significantly by optimizing the element kernel. However, care should be taken to ensure that the element kernel uses the least amount of local memory as possible, as some excess local memory may be allocated for the thread in the off-chip memory space with very high latency.

It can be noted that in some rare instances the running time of the dummy kernels is longer. We could not completely explain this fact. It is most likely caused by a sub-optimal compilation.

In Figure 14 the NVIDIA Tesla C1060 GPU was used. This card has the NVIDIA compute capability v1.3 and an optimized hardware to minimize the number of memory transactions with more aggressive coalescing (see [14]). Notice that for the low order elements, the SharedNZ speedup is now approximately 65x with respect to the host implementation, a speedup of almost 2x to the 8800 GTX. Also, notice that the element computation stage of the GlobalNZ algorithm is a significantly larger portion of the total assembly time. When using the dummy

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using nmeauth.cls

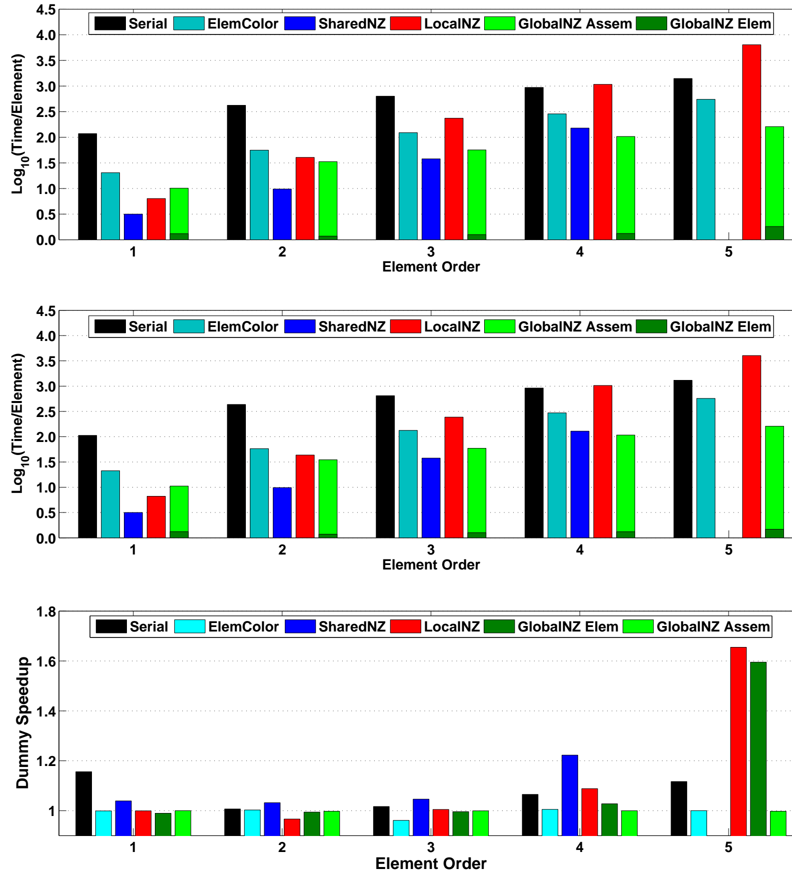


Figure 13. Running time versus order of the elements using the 8800 GTX. (top) The complete finite element assembly operation. (middle) The same assembly run with dummy element kernels to determine an approximate lower bound on the running time due to memory transactions. (bottom) The speedup obtained using the dummy element kernel (note that the y scale does not start at 0).

element subroutines, little or no speedup was found for the low order elements and comparable speedups to the 8800 GTX were found for the higher order ones.

In practice, performance depends on mesh topology and node numbering. Meshes and numbering schemes that improve data locality for the device kernels may reduce the number of memory transactions or increase the number of retrievals from cache.

7. Conclusion

In this paper, we presented a number of strategies to perform computation on an unstructured grid using a GPU. The number of optimizations that are possible is large, in some sense exponentially large since the different strategies can be combined in many different ways. To

Copyright © 2000 John Wiley & Sons, Ltd.

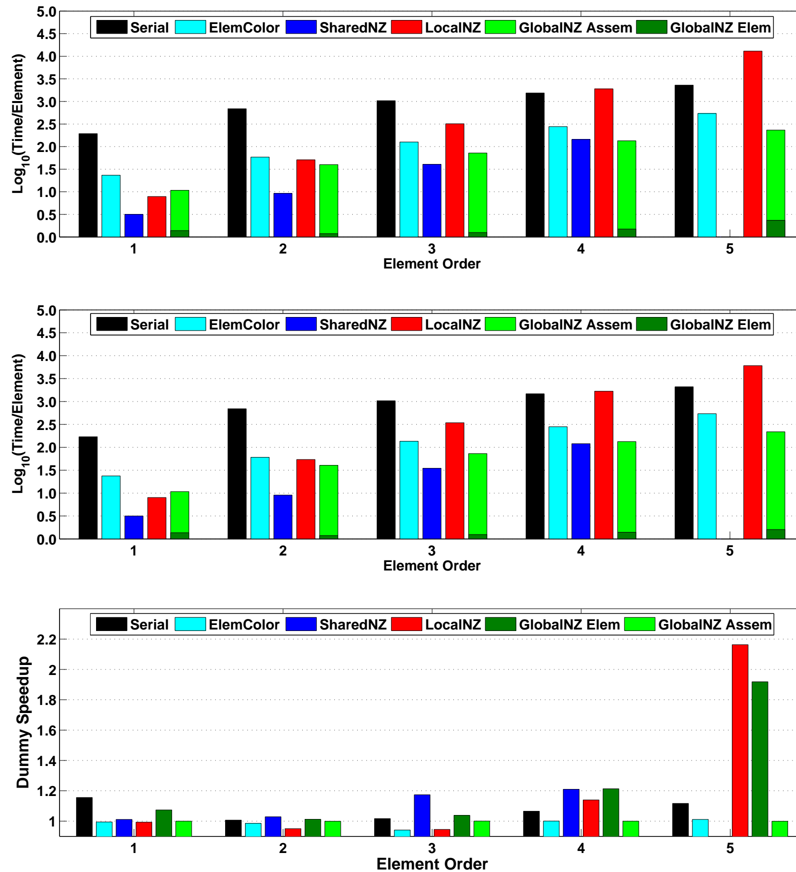


Figure 14. Running time versus order of the elements using the Tesla C1060. (top) The complete finite element assembly operation. (middle) The same assembly run with dummy element kernels to determine an approximate lower bound on the running time due to memory transactions. (bottom) The speedup obtained using the dummy element kernel.

keep the discussion reasonably short, we have presented only the key ideas and what we found to be the best techniques.

For low-order methods or discretizations that result in a small amount of data produced per element, techniques that use shared memory are typically the fastest, as the sharing of information between threads allows reducing the movement of data out of the global memory while not dramatically increasing the number of flops. This is, therefore, an optimal trade-off.

For high-order methods, one typically runs out of shared memory space and the hardware can no longer be used efficiently. Simpler methods are then able to out-perform the shared memory approach. The fastest one uses global memory to store intermediate information in the calculation, before computing the final reduction. Even though this requires an extra pass through global memory, the number of flops the algorithm performs is minimal and overall this is a winning strategy for high-order elements.

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using nmeauth.cls

Komatitsch et al. in [11] present a high order finite element assembly method for CUDA-enabled GPUs based on a coloring of the elements. In the paper, the elements are high order elements with 125 nodes each. The element kernels are parallelized and designed to assign one thread to one node of the element. Although the element coloring approach is straightforward and successfully achieves a significant speedup in [11], the results of this paper suggest that a method that shares nodal data inputs and coalesces outputs for later reductions may be beneficial. Because the elements in [11] are high order with 125 nodes, many optimizations, such as special treatment of interior nodes as mentioned in section 4.2.4, can be devised.

In addition to any speedups obtained by solving on the GPU, a common bottleneck of GPU applications is avoided by reducing the number of data transfers between the host device and the GPU coprocessor. For example, if the GPU is used for data visualization as the solution is determined, very few data transfers are needed. Thus, the assembly, solution, and visualization of a dynamic FEM problem can be performed completely on the GPU. This strategy has already been employed in [12] and [9] with impressive results. This can be especially interesting for real-time visualization either for graphics, design software, gaming ...

This paper has not discussed issues related to clusters of GPUs, even though these are important. Some applications require large scale clusters. Showing that GPUs can scale to multiple nodes without being slowed down significantly by the network latency is critical to demonstrate the usefulness of GPU technology for high-performance large scale parallel computing.

Acknowledgments. This work was partially supported by a research grant from the Academic Excellence Alliance program between King Abdullah University of Science and Technology (KAUST) and Stanford University.

REFERENCES

1. Vogt L, Olivares-Amaya R, Kermes S, Shao Y, Amador-Bedolla C, Aspuru-Guzik A. Accelerating resolution-of-the-identity second-order Møller-Plesset quantum chemistry calculations with graphical processing units. *J. Phys. Chem. A* 2008; **112**(10):2049–2057.
2. Anderson JA, Lorenz CD, Travesset A. General purpose molecular dynamics simulations fully implemented on graphics processing units. *J. Comput. Phys.* 2008; **227**(10):5342–5359.
3. Hardy DJ, Stone JE, Schulten K. Multilevel summation of electrostatic potentials using graphics processing units. *Parallel Comput.* 2009; **35**(3):164–177.
4. Elsen E, LeGresley P, Darve E. Large calculation of the flow over a hypersonic vehicle using a GPU. *J. Comput. Phys.* 2008; **227**(24):10 148–10 161.
5. Göddeke D, Buijssen SH, Wobker H, Turek S. GPU acceleration of an unmodified parallel finite element Navier-Stokes solver. *High Performance Computing & Simulation 2009*, Smari WW, McIntire JP (eds.), 2009; 12–21.
6. Rodriguez-Navarro J, Susin A. Non structured meshes for cloth GPU simulation using FEM. *VRIPHYS* 2006; :1–7.
7. Göddeke D, Strzodka R, Turek S. Accelerating double precision FEM simulations with GPUs. *Proceedings of ASIM 2005*, 2005.
8. Göddeke D, Strzodka R, Mohd-Yusof J, McCormick P, Wobker H, Becker C, Turek S. Using GPUs to improve multigrid solver performance on a cluster. *Int. J. Comput. Sci. Eng.* 2008; **4**(1):36–55.
9. Bolz J, Farmer I, Grinspun E, Schröder P. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. *ACM Transactions on Graphics* 2003; **22**:917–924.
10. Klöckner A, Warburton T, Bridge J, Hesthaven JS. Nodal discontinuous galerkin methods on graphics processors 2009.
11. Komatitsch D, Micha D, Erlebacher G. Porting a high-order finite-element earthquake modeling application to NVIDIA graphics cards using CUDA. *J. Parallel Distr. Com.* 2009; **69**(5):451–460.

Copyright © 2000 John Wiley & Sons, Ltd.

12. Tejada E, Ertl T. Large steps in GPU-based deformable bodies simulation. *Simul. Model. Pract. Th.* 2005; **13**(8):703–715.
13. Natarajan R. Finite element applications on a shared-memory multiprocessor: Algorithms and experimental results. *J. Comput. Phys.* 1991; **94**(2):352–381.
14. NVIDIA Corporation. *NVIDIA CUDA Programming Guide 2.0.* 2008.
15. Volkov V, Demmel J. LU, QR and Cholesky factorizations using vector capabilities of GPUs. *Technical Report UCB/EECS-2008-49*, EECS Department, UC Berkeley May 2008.
16. Brezzi F, Douglas J Jr, Marini LD. Two families of mixed finite elements for second order elliptic problems. *Numerische Mathematik* 1985; **47**:217–235.
17. Kubale M. *Graph Colorings*. American Mathematical Society, 2004.
18. Karypis G, Kumar V. *MeTiS 4.0: Unstructured Graph Partitioning and Sparse Matrix Ordering System* 1998.
19. Graham R. Bounds on multiprocessing timing anomalies. *SIAM J. Appl. Math* 1969; **17**:263–269.
20. Galoppo N, Govindaraju N, Henson M, Manocha D. LU-GPU: efficient algorithms for solving dense linear systems on graphics hardware. *Proceedings of the ACM/IEEE SC 2005 Conference*, 2005; 3–3.
21. Strzodka R, Göddeke D. Pipelined mixed precision algorithms on FPGAs for fast and accurate PDE solvers from low precision components. *IEEE Symposium on Field-Programmable Custom Computing Machines 2006*, 2006; 259–268.
22. Buatois L, Caumon G, Lévy B. Concurrent number cruncher: An efficient sparse linear solver on the GPU. *HPCC*, 2007.
23. Bell N, Garland M. Efficient sparse matrix-vector multiplication on CUDA. *NVIDIA Tech. Report* Dec 2008; **NVR-2008-004**.
24. Baskaran M, Bordawekar R. Optimizing sparse matrix-vector multiplication on GPUs using compile-time and run-time strategies. *Technical Report*, Research Report RC24704, IBM TJ Watson Research Center Dec 2008.
25. He B, Govindaraju NK, Luo Q, Smith B. Efficient gather and scatter operations on graphics processors. *Proceedings of the 2007 ACM/IEEE SC Conference*, 2007; 1–12.

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using nmeauth.cls