Introduction to assembly of finite element methods on graphics processors

2010 IOP Conf. Ser.: Mater. Sci. Eng. 10 012009

(http://iopscience.iop.org/1757-899X/10/1/012009)

IOP Conf. Series: Materials Science and Engineering **10** (2010) 012009 doi:10.1088/1757-899X/10/1/012009

Introduction to Assembly of Finite Element Methods on Graphics Processors

Cristopher Cecka, Adrian Lew, and Eric Darve
496 Lomita Mall, Stanford CA 94305, USA
E-mail: ccecka@stanford.edu

Abstract. Recently, graphics processing units (GPUs) have had great success in accelerating 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 30x or more in comparison to a well optimized serial implementation on the CPU. We also find that the optimal assembly strategy depends on the order of polynomials used in the finite-element discretization.

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 this paper, we only present an introduction to the methods devised and the results obtained. Additional details will be presented in future publications [7]. In that paper, we investigate the use of a single GPU to accelerate the assembly and solution of general finite element methods (FEMs) on unstructured meshes. The approaches discussed 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 can be extended almost unchanged to a three-dimensional model.

Previous studies on FEM in GPUs have largely focused on the solution of the sparse linear system of equations resulting from a FEM discretization [8, 9, 10], mainly because this is often the most computationally demanding step. Other studies have investigated specific unstructured

problems [10, 6, 11, 12] and developed a number of successful methods. However, these methods are often fine tuned to the specific problem, generalize poorly or not at all, and some use outdated hardware and compute environments. A generalized version of the method in [11] is presented in this paper (GlobalNZ), and algorithmic ideas for shared memory machines presented in [13] have been considered and extended for this paper.

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].

2.2. Execution Model

NVIDIA's Compute Unified Device Architecture (CUDA) parallel programming model provides a C/C++ language interface to the hardware [14]. 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 register space. Off-chip global memory is accessible by all threads.

3. Computations on Unstructured Meshes

3.1. Finite Element Method

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

(i) The nodal data matrix $C(n)$, which yields the field values of the n^{th} node, with n being 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 and nodal data will also include the values of all degrees of freedom, such as temperature or displacement.

(ii) The connectivity matrix $E(e, a)$, which yields the global node number of the a^{th} node of the e^{th} element, with a being 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 for non-linear problems, 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.

Alternatively, the algorithms presented 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, as advocated in [15].

3.2. Sequential FEM Assembly

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, for each node in the element. These values are generally retrieved from memory and arranged following a local node numbering scheme for the element. 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 d^{th} degree of freedom of the e^{th} element, $L(e,d)$ is the corresponding global degree-of-freedom number. Using $L(e,d)$ to make the element data dense, an implementation is given in Algorithm 1. This is known as the direct stiffness method and is the most common implementation of a system assembly in the finite element method.

```
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 d_1 of e do
5 |   |   F(L(e,d_1)) += F^e(d_1);
6 |   |   for all local degrees of freedom d_2 of e do
7 |   |   |   A(L(e,d_1),L(e,d_2)) += A^e(d_1,d_2);
```

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

3.3. 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. With very few exceptions, no interblock communication is available in the GPU execution model. Race conditions should be avoided. (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) Optimizing 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.

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. 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].

Although determining the minimum coloring of a general graph is known to be an NP-complete problem, there exist many heuristics for finding nearly minimal colorings [16]. Additionally, we have shown 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.

Let $\mathcal{E}_k$ be the set of elements colored with color k. Precompute the colored element matrices,

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

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 below, which is run for each color k. The global thread ID number is the index of the colored element in E_k being computed.

```
1 tid ← globalThreadID ;        /* tid is the index of a colored element in E_k */
2 for all n ∈ [1, e_n] do
3 ⌊ nodes[n] ← C(E_k(tid, n));
4 Accumulate (A^e, F^e) = elem(nodes) into the system of equations;
```

Algorithm 2: The colored element assembly kernel for finite element assembly.

4.2. Assembly by Non-Zero Entries Using Global Memory

Another approach 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.

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 k^{th} thread block is responsible for computing, then we compute the set of nodes $\mathcal{N}_k$,

$$\mathcal{N}_k = \{E(e, a) : e \in \mathcal{E}_k,\ a = 1, \ldots, e_n\}, \tag{2}$$

where e_n is the number of nodes per element, 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 element subroutines in computing the element data for $\mathcal{E}_k$.

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^ℓ_k, defined by

$$E_k(e, a) = \sigma_k(E(e, a)), \quad \forall\ e \in \mathcal{E}_k,\ a = 1, \ldots, e_n \tag{3}$$

where $\sigma_k : \mathcal{N}_k \to \{1, \ldots, |\mathcal{N}_k|\}$ 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. 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. The resulting algorithm for the first stage of the global assembly method is sketched in Algorithm 3 and diagrammed in Figure 1.

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.

```
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. Here, $end_k = blockSize\ \lceil |\mathcal{E}_k|/blockSize \rceil$, $sMem$ denotes the shared memory space and $gMem$ denotes the global memory space. All reads from the block local element matrix are coalesced. The element subroutine is responsible for writing to global memory with coalesced transactions.

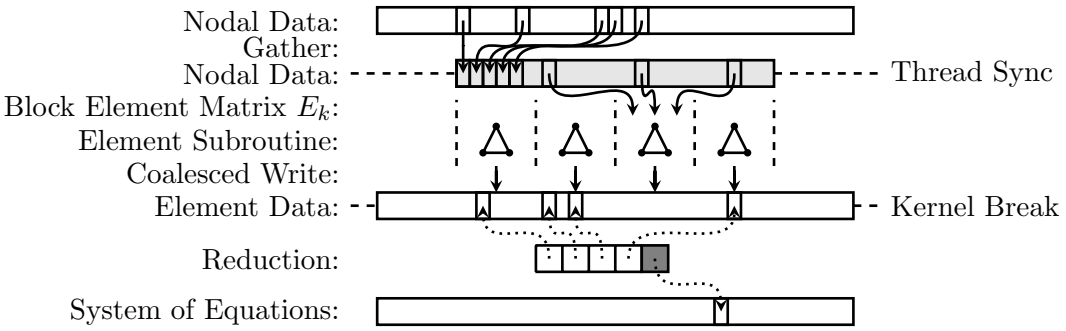


Figure 1. The global assembly by NZ on the GPU. Global memory is depicted in white and shared memory is depicted in light gray. Solid black arrows represent memory reads and writes and dotted arrows represent references to memory. The kernel break denotes the end of the element computation kernel and the beginning of the assembly kernel.

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. That is, the element data is stored in a memory space that has very fast reads and writes and it can be shared across threads in order to reduce the total number of calls to the element subroutine.

First, because there can be no interblock communication between threads, we partition the nodes so that each block of threads is responsible for assembling the degrees of freedom associated with a set of nodes of the mesh. If $\mathcal{N}_k$ is a partition of the nodes, we must find the set of elements

$$\mathcal{E}_k = \{e \in \mathcal{E} \mid \exists\, a \text{ such that } E(e,a) \in \mathcal{N}_k\}, \tag{4}$$

that is, the set of elements adjacent to any node of $\mathcal{N}_k$. As before, the elements of $\mathcal{E}_k$ share many nodes and an important optimization is to minimize the transactions with global memory.

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 thread block. Each thread block k needs to retrieve the set of nodes

$$\overline{\mathcal{N}}_k = \{E(e,a) : e \in \mathcal{E}_k,\ a = 1, \ldots, e_n\}, \tag{5}$$

Figure 2 shows an example of the sets $\mathcal{N}_k$, $\overline{\mathcal{N}}_k$ and $\mathcal{E}_k$.

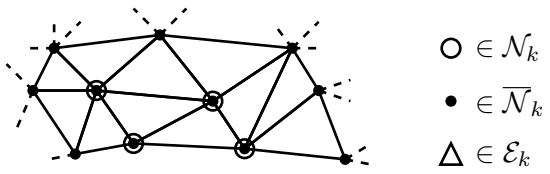
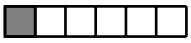


Figure 2. Example of the sets $\overline{\mathcal{N}}_k$, $\mathcal{N}_k$, and $\mathcal{E}_k$. As $|\mathcal{E}_k|$ increases, the ratio $\left|\overline{\mathcal{N}}_k\right| / |\mathcal{N}_k|$ decreases, which improves the efficiency of the algorithm.

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. For each node we create a list of the form



where the dark entry represents the node number (source index) to be retrieved and the light entries represent the block element number that requires the data from that node (target index).

The lists associated to different nodes in $\overline{\mathcal{N}}_k$ may have significantly different lengths. We pack these lists into a *scatter array* using a packing algorithm such as LPT [17]. The result is a column-major matrix with *blockSize* rows and data profile shown in Figure 3. The parallel algorithm to perform the instructions in the scatter array is then particularly simple and is shown in Algorithm 4.

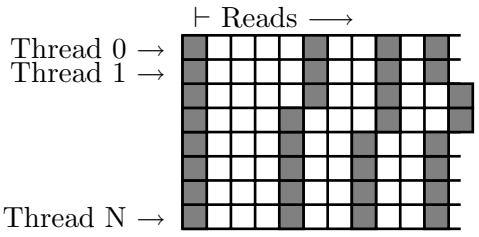


Figure 3. The column-major scatter matrix with *blockSize* rows. The dark entries represent source indices and the light entries represent target indices. A thread block will read a column of the array in coalesced memory transactions.

After the element subroutines have calculated the element data and stored it into shared memory, overwriting the nodal data, we need to 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 and we can use a similar approach to the scatter operation to perform the reduction operation.

The entire procedure is diagrammed in Figure 4. Note that the shared assembly by NZ algorithm is heavily constrained by the size of the shared memory space.

```
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 4: 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.

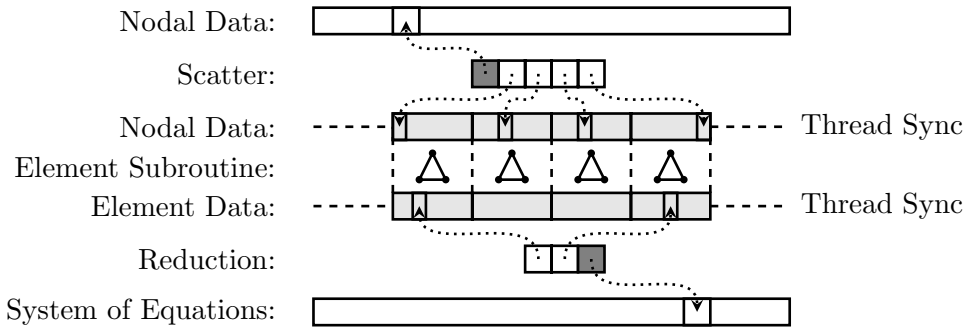


Figure 4. The shared assembly by NZ on the GPU. Global memory is depicted in white and shared memory is depicted in light gray. Dotted arrows represent references into memory.

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. 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 partitioning, prefetching of shared nodal data, and packing NZ dependency lists into a coalesced matrix. Although the amount of redundant data traffic 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 [10].

5. Experimental Results

5.1. 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

$$\begin{aligned} \nabla \cdot (\boldsymbol{\kappa} \cdot \nabla u(\boldsymbol{x})) &= f(\boldsymbol{x}) \quad \boldsymbol{x} \in \Omega \\ u(\boldsymbol{x}) &= g(\boldsymbol{x}) \quad \boldsymbol{x} \in \Gamma_g \\ \boldsymbol{n}(\boldsymbol{x}) \cdot \boldsymbol{\kappa} \cdot \nabla u(\boldsymbol{x}) &= q(\boldsymbol{x}) \quad \boldsymbol{x} \in \Gamma_q \end{aligned} \tag{6}$$

on a the domain pictured in Figure 5 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.

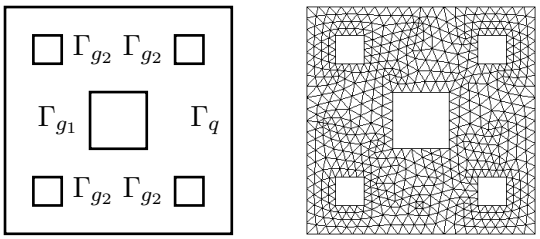


Figure 5. The domain of the finite element test problem and an example mesh.

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. We find that 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 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 $1/h$.

5.2. Element Kernels

For our test case, we write element kernels to compute

$$A^{\mathrm{e}}_{ij} = \int_{\Omega^{\mathrm{e}}} \boldsymbol{\kappa} \nabla \varphi_i \cdot \nabla \varphi_j \, d\Omega \qquad F^{\mathrm{e}}_i = \int_{\Omega^{\mathrm{e}}} \varphi_i f \, d\Omega - \sum_j A^{\mathrm{e}}_{ij} g_j \qquad g_j = \begin{cases} 0 & \text{if } \boldsymbol{x}_j \in \Omega \\ g(\boldsymbol{x}_j) & \text{if } \boldsymbol{x}_j \in \Gamma_g \end{cases}$$

For this paper, we used N^{th} order Lagrange triangular elements. 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 in source code.

The element kernels were written in a modular fashion and are therefore almost identical for all methods. The only differences are in input and output. 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.

It is important to minimize the memory usage of the element kernel. As we will see in section 5.3, 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.

5.3. 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 the direct stiffness method. Optimizations were made to prevent superfluous memory accesses to achieve a speedup of

approximately 2x over a conventional implementation. The reference code is compiled with the `-O2` flag.

The running times of the more successful GPU algorithms are shown in Figure 6 and Figure 7. 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 described in Section 5.1.

The relative running times of the algorithms are presented as a function of the mesh size in Figure 6. 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.

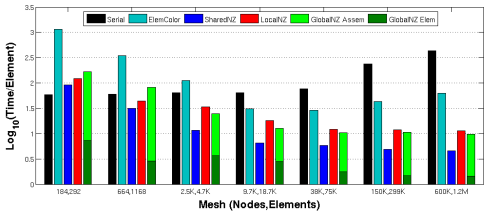


Figure 6. Running time versus size of the mesh.

Figure 7a 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 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.

In Figure 7b 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.

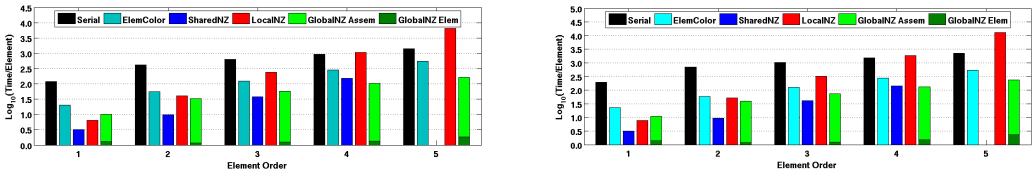


Figure 7. Running time versus order of the elements using the 8800 GTX and C1060.

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.

6. Conclusion

In this paper, we presented a number of strategies to perform computation on an unstructured grid using a GPU. To 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.

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.

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. 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 [10] with impressive results. This can be especially interesting for real-time visualization either for graphics, design software, and gaming.

References

[1] Vogt L, Olivares-Amaya R, Kermes S, Shao Y, Amador-Bedolla C and Aspuru-Guzik A 2008 *J. Phys. Chem. A* **112** 2049–2057
[2] Anderson J A, Lorenz C D and Travesset A 2008 *J. Comput. Phys.* **227** 5342–5359
[3] Hardy D J, Stone J E and Schulten K 2009 *Parallel Comput.* **35** 164–177
[4] Elsen E, LeGresley P and Darve E 2008 *J. Comput. Phys.* **227** 10148–10161
[5] Göddeke D, Buijssen S H, Wobker H and Turek S 2009 *High Performance Computing & Simulation 2009* ed Smari W W and McIntire J P pp 12–21
[6] Rodriguez-Navarro J and Susin A 2006 *VRIPHYS* 1–7
[7] Cecka C, Lew A and Darve E *Submitted to Int. J. for Num. Methods in Eng.*
[8] Göddeke D, Strzodka R and Turek S 2005 *Proceedings of ASIM 2005*
[9] Göddeke D, Strzodka R, Mohd-Yusof J, McCormick P, Wobker H, Becker C and Turek S 2008 *Int. J. Comput. Sci. Eng.* **4** 36–55
[10] Bolz J, Farmer I, Grinspun E and Schröder P 2003 *ACM Transactions on Graphics* **22** 917–924
[11] Komatitsch D, Micha D and Erlebacher G 2009 *J. Parallel Distr. Com.* **69** 451–460
[12] Tejada E and Ertl T 2005 *Simul. Model. Pract. Th.* **13** 703–715
[13] Natarajan R 1991 *J. Comput. Phys.* **94** 352–381
[14] NVIDIA Corporation 2008 *NVIDIA CUDA Programming Guide 2.0*
[15] Rumpf M and Strzodka R 2005 *Numerical Solution of Partial Differential Equations on Parallel Computers* (*Lecture Notes in Comp. Sci. and Eng.* vol 51) ed Bruaset A M and Tveito A (Springer) pp 89–134
[16] Kubale M 2004 *Graph Colorings* (American Mathematical Society)
[17] Graham R 1969 *SIAM J. Appl. Math* **17** 263–269