Int J Parallel Prog (2015) 43:203–239
DOI 10.1007/s10766-013-0301-6

Finite Element Algorithms and Data Structures on Graphical Processing Units

I. Z. Reguly · M. B. Giles

Received: 27 March 2013 / Accepted: 23 November 2013 / Published online: 4 December 2013

Abstract The finite element method (FEM) is one of the most commonly used techniques for the solution of partial differential equations on unstructured meshes. This paper discusses both the assembly and the solution phases of the FEM with special attention to the balance of computation and data movement. We present a GPU assembly algorithm that scales to arbitrary degree polynomials used as basis functions, at the expense of redundant computations. We show how the storage of the stiffness matrix affects the performance of both the assembly and the solution. We investigate two approaches: global assembly into the CSR and ELLPACK matrix formats and matrix-free algorithms, and show the trade-off between the amount of indexing data and stiffness data. We discuss the performance of different approaches in light of the implicit caches on Fermi GPUs and show a speedup over a two-socket 12-core CPU of up to 10 times in the assembly and up to 6 times in the solution phase. We present our sparse matrix-vector multiplication algorithms that are part of a conjugate gradient iteration and show that a matrix-free approach may be up to two times faster than global assembly approaches and up to 4 times faster than NVIDIA's cuSPARSE library, depending on the preconditioner used.

Keywords Graphical processing unit · Finite element method · Performance analysis · Sparse matrix-vector multiplication · Preconditioned conjugate gradient method

I. Z. Reguly
Pázmány Péter Catholic University, Práter u. 50/a, Budapest 1083, Hungary

I. Z. Reguly (✉) · M. B. Giles
Oxford e-Research Centre, 7 Keble Road, Oxford OX1 3QG, UK
e-mail: istvan.reguly@oerc.ox.ac.uk

M. B. Giles
e-mail: mike.giles@maths.ox.ac.uk

1 Introduction

Due to the physical limitations to building faster single core microprocessors, the development and use of multi- and manycore architectures has received increasing attention for the past few years. Besides the increasing number of processor cores on a single chip, new architectures have emerged that support general purpose massively parallel computing—the most prominent of which are Graphical Processing Units (GPUs). Computing on Graphical Processors has become very popular in the high performance computing community; a great number of papers discuss its viability in accelerating applications ranging from molecular dynamics through dense and sparse linear algebra to medical imaging [14].

The evolution trend of computing architectures shows an increasing gap between computational performance and bandwidth to off-chip memory [8]; it is already apparent that while GPUs have a theoretical compute performance ten times higher than CPUs, the bandwidth to off-chip DRAM is only 3–5 times faster. Also, with an increasing number of cores on a single chip the amount of on-chip memory per core decreases. Thus, it is becoming increasingly important to move as little data as possible—sometimes even at the expense of redundant computations.

The numerical approximation of Partial Differential Equations (PDEs) can be split into two main categories: structured and unstructured grids. Finite difference methods are well suited for structured grids; their data access patterns are regular and easily vectorisable. These methods are usually described as applying stencil operations to the elements of a structured grid. This approach to solving PDEs is a natural fit for GPUs, and several studies have shown considerable speedup over multicore CPUs [7,9]. However, in practice the domain over which PDEs are defined is often complex and requires numerical accuracy in some areas more than other, which may require the use of unstructured grids. For the solution of these problems, the finite volume and the finite element methods are more convenient [15]. When using these unstructured grids, memory access patterns become complicated. It usually involves gathering input data and scattering output data, which results in having to deal with race conditions on parallel hardware.

2 Related Work

Due to the high demand for accelerating finite element methods (FEM) several studies have investigated FEM implementations on GPUs. The method can be divided into two phases: the assembly of a matrix, then the solution of a linear system using that matrix. In general the solution phase is the more time consuming one, specifically the sparse matrix-vector product between the assembled matrix and vectors used by iterative solvers. Because this is a more general problem, several studies focused on the performance evaluation of this operation on the GPU [3,4,27] and in the FEM context [13]. Finite element assembly has also been investigated both in special cases [4,12,16,17] and in more general cases to show the alternative approaches to matrix assembly for more general problems [6,11,18,23]. Their results show a speedup of 10 to 50 compared to single threaded CPU performance in the assembly phase, but only very limited speedup in the iterative solution phase.

The goal of this paper is to explore different options for assembling and solving finite element problems on unstructured grids. Our hand-written tests focus on using the Compute Unified Device Architecture (CUDA) on NVIDIA GPUs, and OpenMP on Intel CPUs. Several aspects are explored that impact the performance of these algorithms such as memory layout of input and output data, the trade-off between computation and storage, preprocessing, and autotuning. The effects of these on performance are discussed and recommendations are made. To our knowledge, this paper is one of the most comprehensive studies of finite element algorithms that consider different memory layouts and bottlenecks including implicit caches that were introduced with NVIDIA's Fermi architecture.

The contributions of this paper are the following:

1. Presents the trade-offs between computation, data storage and movement, develops an assembly algorithm that scales to high degree polynomials and discusses data transfer requirements for different storage approaches.
2. Presents and discusses the local and global matrix assembly approaches using several storage formats, evaluates and compares their performance revealing the bottlenecks of the assembly, iterative solution and preconditioning phases of the calculation.
3. Presents and discusses GPU specific techniques to avoid race conditions, makes use of caching by preprocessing techniques, and optimises occupancy with autotuning.

The paper is organised as follows: Sect. 3 discusses the essential mathematical background of the finite element method that is referred to throughout the paper. Section 4 briefly discusses the different aspects of GPU programming, presents the algorithm for finite element assembly that is used throughout the tests, and also introduces the sparse matrix storage formats used. In Sect. 5 we present the test hardware and problem, and briefly describe the different test cases. Sections 6, 7, 8, and 9 present and discuss the different versions for global and local matrix assembly. Sections 10 and 11 perform the comparative analysis of different approaches and assesses the performance bottlenecks of each phase in the finite element method. Finally, Sect. 12 draws conclusions.

3 The Finite Element Method on Unstructured Meshes

Unstructured meshes are used in many engineering applications as a basis for the discretised solution of PDEs, where the domain itself or the required accuracy of the solution is nonuniform. Variations of the Finite Element Method (FEM) can handle most types of partial differential equations, but to understand the algorithm, consider the following simple boundary value problem over the domain Ω [15]:

$$-\nabla \cdot (\kappa \nabla u) = f \text{ in } \Omega, \tag{1}$$

$$u = 0 \text{ on } \partial\Omega. \tag{2}$$

The solution is sought in the form of $u : \Omega \rightarrow \Re$. With the introduction of a test function v on Ω, the *variational form* of the PDE is as follows:

$$\text{Find } u \in V \text{ such that } \int\limits_{\Omega} \kappa \nabla u \cdot \nabla v \, dV = \int\limits_{\Omega} f v \, dV, \ \forall v \in V, \tag{3}$$

where V is the finite element space of functions which are zero on $\partial\Omega$, or as reformulated below:

$$u : \ a(u, v) = \ell(v), \ \forall v \in V, \tag{4}$$

$$a(u, v) = \int\limits_{\Omega} \kappa \nabla u \cdot \nabla v \, dV, \tag{5}$$

$$\ell(v) = \int\limits_{\Omega} f v \, dV. \tag{6}$$

The standard finite element method constructs a finite dimensional space $V_h \subset V$ of functions over Ω, and searches for an approximate solution $u_h \in V_h$. Let $\{\phi_{1 \ldots N_v}\}$ be a basis for V_h, then:

$$u_h = \sum_i \bar{u_i} \phi_i \tag{7}$$

To find the best approximation to u, it is necessary to solve the system:

$$K \bar{u} = \bar{l}, \tag{8}$$

where K is the $n \times n$ matrix, usually called the *stiffness matrix*, defined by:

$$K_{ij} = a(\phi_i, \phi_j), \ \forall i, j = 1, 2, \ldots, N_v, \tag{9}$$

and $\bar{l} \in \Re^n$, usually called the *load vector*, is defined by:

$$\bar{l}_i = \ell(\phi_i), \ \forall i = 1, 2, \ldots, N_v. \tag{10}$$

If the underlying discretisation mesh has nodes $\bar{x}_i$, it is possible to choose a finite element space V_h with basis functions such that $\phi_i(\bar{x}_j) = \delta_{i,j}$. In this case u_h is determined by its values at $\bar{x}_i, \ i = 1, 2, \ldots N_v$. The mesh is a polygonal partitioning of the domain Ω into a set of disjoint elements $e_i \in E, \ i = 1 \ldots N_e$. The basis functions are constructed so that ϕ_i is nonzero only over those elements e which have $\bar{x}_i$ as a vertex. This means that finite element basis functions ϕ_i have their support restricted to neighbouring elements.

The essential data to describe the mesh is as follows:

1. Global index for each node in the mesh $I = \{1 \ldots N_v\}$.
2. Coordinate data for each node in the mesh $\bar{x}_i \in \Re^d$.
3. List of elements $E = \{1 \ldots N_e\}$;

4. Element → node mapping M_e mapping from elements of E to a subset of I, where n is the number of *degrees of freedom—d.o.f.* in an element. This mapping is an ordered list of global node indices, for example in a counter-clockwise fashion.

3.1 Finite Element Assembly

Because the basis functions ϕ_i have their support restricted to neighboring nodes, the bilinear form in (5) can be partitioned and constrained to be the sum of integrals over a few elements. In practice the stiffness matrix can be constructed by iterating through every element and adding up the contributions from integrals of nonzero basis functions $\phi_i,\ i \in M_e(e)$ over the current element Ω_e, as shown in Algorithm 1.

Algorithm 1 Element by element assembly of the stiffness matrix and the load vector

```
for each element $e \in E$ do
  for each degree of freedom $i \in M_e(e)$ do
    for each degree of freedom $j \in M_e(e)$ do
      $K_{ij} += \int_{\Omega_e} \kappa \nabla\phi_i \cdot \nabla\phi_j \, dV$
    end for
    $\bar{l}_i += \int_{\Omega_e} f\phi_i \, dV$
  end for
end for
```

The resulting matrix K will be sparse because only neighboring basis functions' products will be nonzero. The bilinear form $a(.,.)$ is symmetric, so K is symmetric too. This algorithm can be decribed in another way; assembling dense *local matrices* K_e which contain the nonzeros that the current element contributes to the global matrix, then scattering these values to their place in the global matrix.

3.2 Dirichlet Boundary Conditions

In general, essential boundary conditions constrain the solution u:

$$u = g \text{ on } \Gamma_{Dirichlet} \subset \partial\Omega, \tag{11}$$

for some function g on the constrained part of the boundary $\partial\Omega$. This means the nodes on the boundary have fixed values. This change of course has to appear in the linear system $K\bar{u} = \bar{f}$. In our experiments we assume $g = 0$ and $\Gamma = \partial\Omega$ according to Eq. (2). There are two popular ways to implement this, either via pre- or post-processing:

1. Before assembly, renumber the nodes of the mesh in a way that constrained nodes are not included in the linear system, thereby eliminating them from further computations at the expense of having to look up different indices for accessing data on the nodes and accessing the linear system.

2. Assemble the stiffness matrix and load vector as if no nodes were constrained, then set each constrained node's row and column to zero, the diagonal element to one, and the entry in the right hand side vector to the prescribed value.

Our test programs use the first approach, the non-constrained nodes are referred to as *free nodes* or *degrees of freedom* $I_f = \{1 \ldots N_f\}, \ N_f \leq N_v$.

3.3 The Local Matrix Approach (LMA)

The previous sections described the assembly of the stiffness matrix by scattering the values of the element matrices. There are many numerical methods to approximate the solution of the system of equations $K\bar{u} = \bar{l}$; many of them, such as the *conjugate gradient method*, only perform matrix-vector products and do not explicitly require the matrix K. In the $\bar{y} = K\bar{x}$ multiplication the *local matrix approach* gathers the values of $\bar{x}$, performs the multiplication with the local matrices, and finally scatters the product values to $\bar{y}$ [5,18]. Formally this can be described as follows:

$$\bar{y} = \mathcal{A}^T(K_e(\mathcal{A}\bar{x})), \tag{12}$$

where K_e is the matrix containing the local matrices in its diagonal and $\mathcal{A}$ is the local-to-global mapping from the local matrix indices to the global matrix indices. Formally, this requires three sparse matrix-vector products, but as it is shown in Sect. 8 the mapping matrix $\mathcal{A}$ does not have to be constructed, and the whole operation reduces to N_e dense matrix—vector products, the gathering of $\bar{x}$, and the scattering of products to $\bar{y}$.

3.4 The Matrix-Free Approach

The logic of the local matrix approach can be taken a step further; never writing local stiffness matrices to memory but recalculating them every time the matrix-vector product is performed. This approach can save a high amount of memory space and bandwidth by not moving stiffness data to and from memory. This method has a different data transfer versus computation ratio than the others, thus makes an interesting case when exploring performance bottlenecks of the finite element algorithm.

4 Implementation Considerations for GPUs

GPUs are naturally parallel architectures with their own performance considerations:

1. On NVIDIA GPUs, groups of 32 threads called warps are executed in a single instruction multiple data (SIMD) fashion. A collection of threads called a thread block is assigned to a Streaming Multiprocessor (SM); each SM can process up to 8 thread blocks at the same time, but no more than 1,536 threads (for compute capability 2.0). If there are too many thread blocks, then some execute only after others have finished.

2. The problem has to be decomposed into fairly independent tasks because collaboration is very limited. Threads in a thread block can communicate via on-chip shared memory. For threads that are not in the same thread block, communication is only possible indirectly via expensive operations through global memory, and even then synchronization is only possible by starting a new kernel.
3. Memory bandwidth is very limited between the host and the GPU and has a high latency thus any unnecessary transfers should be avoided.
4. Memory bandwidth to off-chip graphics memory is highly dependent on the pattern of access and type of access. If threads in the same warp access memory locations adjacent to each other, than these memory transfers can be bundled together resulting in a so called *coalesced memory access* and the full width of the memory bus can be utilised. With the introduction of caching, coalesced access is no longer a strict requirement, however, for reasons described below, it is still desirable.
5. Implicit L1 and L2 caches were introduced with the Fermi architecture; each Streaming Multiprocessor (SM) has a 16k/48k L1 cache, and there is a single shared 768k L2 cache for the whole chip. The L1 cache size is comparable to the cache on the CPU (usually 32k), however while a CPU core executes only a few (1–2) threads, the GPU executes up to 1,536 threads per SM. This results in very constrained cache size per thread: since a cache line is 128 bytes long, even when using 48k L1 cache, only 384 lines can be stored. If all threads read or write in a non-coalesced way, only 384 of them can get cache hits when accessing that cache line, the others get a cache miss. Cache hits can greatly improve performance, but misses cause high latency and potentially cache trashing. It is therefore very important for threads in the same block to work on the same cache lines, to have so called “cache locality”.
6. Instruction throughput depends on both the number of threads per SM and the type of instructions: Fermi’s support for Fused Multiply-Add (FMA) enables high floating point throughput, however unstructured grids require a considerable amount of pointer arithmetic for which there are no separate integer units—unlike in CPUs.
7. Precision requirements are linked to all of the above; double precision floating point arithmetic requires more clock cycles for execution, and doubles the bandwidth requirements.

When working on unstructured grids, the biggest problem is the gather-scatter type memory access, and much depends on the ordering of elements and their nodes. In the assembly phase the nodal data (coordinates and state variables) have to be gathered for each degree of freedom in an element. This data is accessed indirectly via the M_e mapping, which usually results in an uncoalesced memory access. After assembling the local matrices, their elements have to be scattered to populate the global stiffness matrix. The latter operation poses a data hazard, as neighbouring elements have to write to overlapping segments of memory in the global matrix. This problem can be solved either by colouring the elements and executing the writes colour by colour [24], or by the use of atomic operations—however these are not yet available for double precision variables. The problem can be avoided altogether by choosing either nonzeros or whole lines of the stiffness matrix as a unit of work to be assigned to threads [6]. These approaches only involve gather type operations, however the amount of computation required and data moved is higher since the algorithm has to iterate through all the elements connected to the given degree of freedom.

4.1 The Finite Element Algorithm on GPUs

The algorithms in this paper are based on quadrilateral elements and can work for elements of any order. The calculation of the coefficients of the basis functions, the local quadrature points and the gradients are based on a transformation from the reference square. This bilinear transformation is calculated for every element and applied to the data of the reference square stored in constant memory. The pseudocode for each element is described by Algorithm 2.

When increasing the degree of polynomials used as basis functions, both the number of degrees of freedom and the number of quadrature points increase as a square function of the degree. For 2D quadrilateral elements it is equal to: $(degree + 1)^2$. To perform the minimal amount of computations, the local quadrature points and the inverse of the jacobian evaluated at each one of them can be precomputed and reused in the innermost loop of Algorithm 2. Alternatively, any of these can be recalculated every time they are needed, thereby saving local storage. This enables us to trade computations for local storage space. In the CPU versions, where register pressure does not pose a problem, all these values are precomputed and contained by on-chip caches. However, the quickly growing register pressure makes GPU implementation infeasible for higher degrees. Thus, we have two implementations for the GPU, one which precomputes the coordinates of local quadrature points and reuses them and one which recalculates them every time they are needed in the innermost loop. The latter kernel does not store anything in local arrays that would grow in size with the increasing degree of polynomials used—thus it uses the same number of registers for any degree.

The third approach to the assembly is to exchange the loops over the quadrature points and the pairs of degrees of freedom, in which case the coordinates of the local quadrature points and the jacobian do not have to be recalculated. However, in this case the values of the local stiffness matrix are updated repeatedly. This is a viable option on the CPU, but on the GPU these local matrices cannot fit in either the local memory or the cache which results in dramatically increased memory traffic.

For testing purposes meshes were generated with different numbers of elements and with different degree elements in a way such that all meshes have approximately the same total number of degrees of freedom. The default numbering of the degrees of freedom is based on iterating through elements and their degrees of freedom and assigning a number in an increasing order. In several scenarios a colouring of these elements [24] is required to avoid write conflicts. An example mesh with seven second order elements and a total number of 39 degrees of freedom is shown in Fig. 1; different colours are indicated with different capital letters.

Colouring is done on two levels: block and element. Since there is no explicit synchronisation between blocks in CUDA, blocks of elements with different colour are processed by different kernel calls, thereby making sure that no two blocks access the same data at the same time. Within these blocks different elements are assigned to each thread along with their colour. When these threads access memory in a poten-

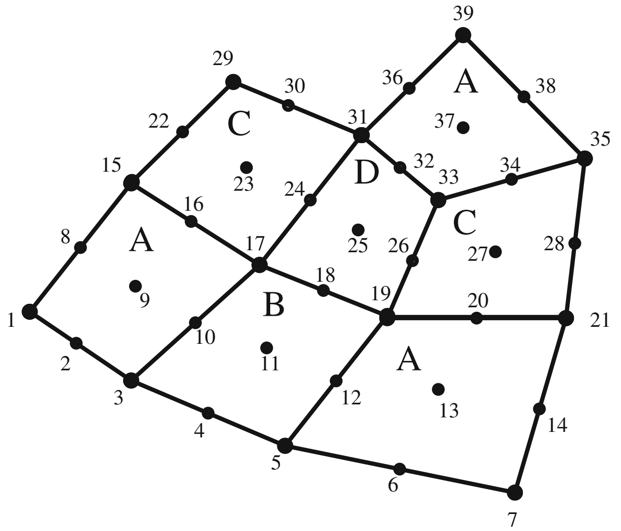


Fig. 1 An example of an unstructured grid along with the numbering of the degrees of freedom and colouring of the elements

tially conflicting way, these accesses are executed colour-by-colour with an explicit synchronisation between each one. The colouring of elements was done by iterating through them and assigning the first colour available that was not used by any of the element's neighbours.

Most GPU algorithms differ only in the way the local stiffness matrix and load vector are written to global memory.

Algorithm 2 Local stiffness matrix and load vector assembly

```
$I_{local} \leftarrow M_e(e)$
generate bilinear $mapping$ based on the four vertices
calculate local quadrature points $quadPoints_{local}$
Calculate jacobians $J$ to map to each quadrature point
for $i = 1$ $to$ $length(I_{local})$ do
  if $i$ is not constrained then
    for $j = i$ $to$ $length(I_{local})$ do
      if $k$ is not constrained then
        for each quadrature point $p$ in $quadPoints_{local}$ do
          $K_{local}(i, j) += weights(p) * \nabla\phi_i(p) * \nabla\phi_j(p)$
        end for
        $K_{local}(i, j) = K_{local}(j, i)$
      end if
    end for
    for each quadrature point $p$ in $quadPoints_{local}$ do
      $\bar{l}_{local}(i) += weights(p) * \phi_i(p) * f(p)$
    end for
  end if
end for
```

4.2 Data Layout

To minimise the amount of data transferred and improve data reuse on the GPU, the access pattern and thus the layout of the data that is accessed is very important. There are two stages of computation that take the most time: the assembly and the sparse matrix-vector product. This section discusses the layout of input data for the assembly phase and the layout of the sparse matrix that is both the output of the assembly and the input of the matrix-vector product phase.

Input mesh data consists of the mapping M_e, the node coordinates, the mapping from nodes to the indices of the linear system and the state variables for each node. Since the algorithm iterates through elements, only the mapping data M_e is accessed directly. To achieve coalesced reads this is arranged so that the global node indices of the nth degree of freedom in each element e are adjacent to each other:

$$index = n * N_e + e. \tag{13}$$

Hence, when a warp of 32 threads loads in the mapping data for the nth d.o.f. of their respective elements, that data is a consecutive block in memory, giving a coalesced transfer.

The rest of the input data in the assembly phase is accessed in an indirect fashion, based on the global indices of the degrees of freedom.

The layout of stiffness matrix data is one of the most important factors that determine the speed of the finite element method on the GPU: in the assembly phase the amount of data written to the stiffness matrix is at least comparable to the amount of input mesh data, but for higher order elements it is many times higher. Since the sparse-Matrix Vector (spMV) product is evaluated repeatedly, the matrix data will be read multiple times and hence it is very important to optimise its access.

There are several different sparse matrix storage formats used on GPUs [3]; this paper uses the CSR and ELLPACK formats in addition to storage according to the local matrix assembly approach described in Sect. 3.3:

1. COO (COOrdinate storage) stores every non-zero, their row, and column indices separately. These vectors are not necessarily ordered, so in the spMV phase this may result in random accesses to the multiplicand and the result vector and in the latter case write conflicts may occur. It is also difficult to populate such a matrix on a parallel architecture. Because of these disadvantages our tests did not use this storage scheme.
2. CSR (Compressed Sparse Row) stores the non-zeros in a row-major order. As shown in Fig. 2, this format uses two arrays storing every non-zero and their column indices, and an additional array with pointers to the first nonzero of every row and the total number of non-zeros at the end. The size of the first two arrays are the same as the number of non-zeros in the matrix, the size of the third array equals the number of rows plus one. The advantage of CSR is that it is very compact and easy to read, however it is very difficult to populate in a dynamic way; preprocessing is necessary to determine the locations of the non-zeros beforehand. This also results in a lot more input data in the assembly phase.

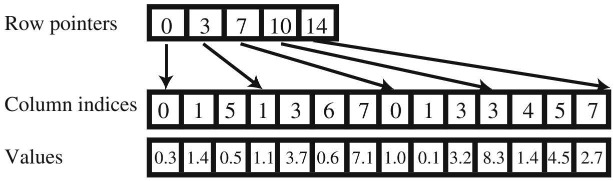


Fig. 2 Memory layout of the Compressed Sparse Row (CSR) format on the GPU

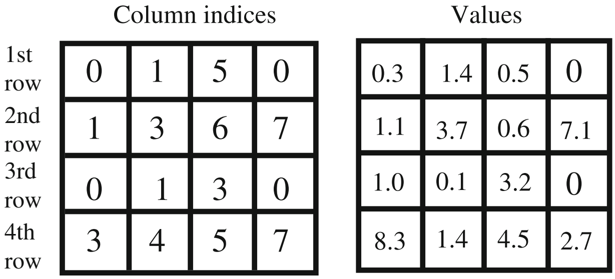
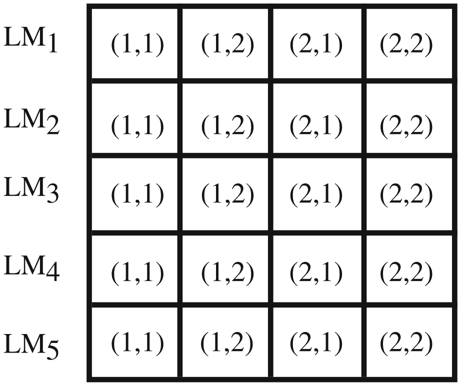


Fig. 3 Memory layout of the ELLPACK format. *Rows* are padded with *zeros*

3. ELLPACK [3,27] stores the sparse matrix in two arrays, one for the values and one for the column indices. Both arrays are of size $number\ of\ rows * max\ row\ length$. Note that the size of all rows in these compressed arrays are the same because every row is padded as shown in Fig. 3. In a standard CPU implementation the data (both indices and values) are stored row-by-row. In our GPU implementation we transpose this, enabling coalesced transfers when threads are assigned to process different rows of the matrix.
4. Hybrid ELLPACK [3] stores the elements up to a certain row length in the ELLPACK format and the rest in a COO format. This layout has smaller memory requirements than the ELLPACK, but it is more difficult to use because of the COO part. Due to the relatively well-known length of the rows in the stiffness matrix, we only use the ELLPACK format.
5. In the LMA approach we store the dense element matrices, which are of size $(\#d.o.f.\ per\ element)^2$. These matrices are stored in vectors, each one containing the local matrix in a row-major order as shown in Fig. 4. These vectors are then stored row-by-row in memory. In a similar way to ELLPACK, on the GPU we transpose this layout to get coalesced memory transfers.

4.3 Estimating Data Transfer Requirements

There is disparity between memory bandwidth and computational power on modern many-core architectures—the Tesla C2070 has a peak floating-point instruction throughput of 1 TFLOPS and a memory bandwidth of 144 GB/s, thus for any single precision number transferred there have to be 27 floating-point instructions to balance data transfer with computation. It is therefore essential to minimise the amount of data transferred, and to optimise the memory access patterns to fully utilise the bandwidth



LM_1	(1,1)	(1,2)	(2,1)	(2,2)
LM_2	(1,1)	(1,2)	(2,1)	(2,2)
LM_3	(1,1)	(1,2)	(2,1)	(2,2)
LM_4	(1,1)	(1,2)	(2,1)	(2,2)
LM_5	(1,1)	(1,2)	(2,1)	(2,2)

Fig. 4 Memory layout of local matrices (LM). Elements are stored in a row vector with row-major ordering. On the GPU, this layout is transposed to enable coalesced memory accesses

available. To provide an estimate of the amount of data to be transferred in the assembly and solution phases, consider a 2D mesh with quadrilateral elements where the average degree of vertices is four, and the number of elements is N_e. Let p denote the degree of polynomials used as basis functions.

The total number of degrees of freedom is the sum of $d.o.f.$s on the vertices, the edges and inside the elements, not counting some on the boundaries, their number is approximately:

$$\begin{aligned} N_{vertex} &= N_e, \\ N_{edge} &= 2 * N_e(p-1), \\ N_{inner} &= N_e(p-1)^2. \end{aligned} \tag{14}$$

In the assembly phase every approach has to read the coordinates of the four vertices, the mapping M_e, write the elements of the stiffness matrix and the load vector. Additionally, global assembly approaches have to read indexing data to determine where to write stiffness values, which involves at least as many reads per element as there are values in the local stiffness matrices. Thus, for every element,

$$T_{assembly,LMA} = 2 * 4 + (p+1)^2 + (p+1)^4 + (p+1)^2, \tag{15}$$

$$T_{assembly,GMA} = T_{assembly,LMA} + (p+1)^4, \tag{16}$$

where T_{LMA}, T_{GMA} denote the units of data transferred per element for the local and the global matrix approaches. It is clear that the local matrix approach moves significantly less data than the global assembly approaches when p is large

In the sparse matrix-vector multiplication the matrix values, row indices, column indices and the values of the multiplicand and result vectors have to be moved. The local matrix approach has to access all of the local stiffness matrices, the mapping M_e to index rows and columns and the corresponding values of the two vectors for every element. Thus, the amount of data moved is:

$$T_{spMV,LMA} = N_e(p+1)^4 + 3 * N_e(p+1)^2. \tag{17}$$

Table 1 Ratio between data moved by global and local matrix approaches during the spMV in single and double precision

Degree	1	2	3	4
ELLPACK/LMA single	1.0	1.81	2.25	2.49
CSR/LMA single	1.04	1.85	2.28	2.51
ELLPACK/LMA double	0.9	1.58	1.93	2.12
CSR/LMA double	0.92	1.6	1.95	2.13

For global assembly approaches, the matrix is already assembled when performing the spMV, the length of any given row depends on whether the $d.o.f.$ corresponding to that row was on a vertex, an edge, or inside an element:

$$\begin{aligned} L_{vertex} &= (2p+1)^2, \\ L_{edge} &= (2p+1)(p+1), \\ L_{inner} &= (p+1)^2, \end{aligned} \tag{18}$$

based on (14), the total number of stiffness values plus the column indices and the values of the multiplicand and result vectors:

$$\begin{aligned} T_{spMV,GMA} = 3 * (&N_{vertex} * L_{vertex} \\ &+ N_{edge} * L_{edge} + N_{inner} * L_{inner}) \\ &+ N_{vertex} + N_{edge} + N_{inner}. \end{aligned} \tag{19}$$

In addition to this, the CSR format has to read indexing data for rows as well, the size of which is $N_{vertex} + N_{edge} + N_{inner}$.

Table 1 shows the relative amount of data moved by global matrix approaches compared to the local matrix approach for different degree of polynomials used at single and double precision. Observe, that based on these calculations it would be always worth doing LMA except for first degree elements at double precision. However these figures do not take the effects of caching nor the atomics/colouring employed by LMA into account, so we expect the actual performance to be somewhat different. Similar calculations can be carried out for the three dimensional case as well, with the global matrix approaches moving less data in both precisions at first degree elements and significantly more at higher degrees.

5 Experimental Results

5.1 Test Problem

Since our goal was to investigate the relative performance of the finite element method using different approaches on different hardware we chose a simple Poisson-problem with a known solution:

$$-\Delta u(\bar{x}) = sin(\pi x_1) \cdot sin(\pi x_2), \tag{20}$$

$$u(\bar{x}) = 0 \; on \; \partial\Omega. \tag{21}$$

The underlying two dimensional grid consists of quadrilateral elements with up to 16 million nodes and the order of elements ranges from 1 to 4. The tests run on grids that have the same number of degrees of freedom, so the number of elements in a fourth degree test case is one sixteenth of the number of elements in a first degree test case. A conjugate gradient iterative method was used to approximate the solution of the linear system $K\bar{u} = \bar{l}$, that is first evaluated without using preconditioners. We analyse the implications of preconditioners separately, by comparing a simple diagonal (Jacobi) and a Symmetric Successive Over-Relaxation (SSOR) type preconditioner.

5.2 Test Hardware and Software Environment

The performance measurements were obtained on a workstation with two Intel Xeon X5650 6-core processors, clocked at 2.67 GHz with 12 Mb shared L3 cache, 256kB L2, and 32kB L1 cache per core, 24 GBytes of system memory running Ubuntu 10.10 with Linux kernel 2.6.35. The system had 2 NVIDIA Tesla C2070 graphical processors installed, both with 6 GB global memory clocked at 1.5 GHz and 364-bit bus width, 448 CUDA cores in 14 streaming multiprocessors (SMs) clocked at 1.15GHz. The CPU codes were compiled with Intel's C Compiler 11.1, using SSE 4.2, Intel's OpenMP library and the *O2* optimisation flag. The GPU codes were compiled with NVIDIA's nvcc compiler with the CUDA 4.2 framework and the $-use\_fast\_math$ flag.

For accurate timing of GPU computations we used CUDA Events as described in Sect. 8.1.2 of the CUDA C Best Practices Guide [20]. CPU timings were obtained by using the $clock\_gettime$(*CLOCK_MONOTONIC*) function in the standard linux $time.h$ header, called outside of any OpenMP parallel region.

5.3 Test Types

Several tests were performed that aim at investigating different aspects of the finite element algorithm. These tests are analysed from the viewpoint of different performance metrics such as speed and limiting factors, and also their place in the whole algorithm.

1. Stiffness matrix assembly: different approaches to assembly were tested, using the CSR and the ELLPACK sparse storage format, the LMA and the Matrix-free approach. Assembly approaches trading off computations for communications (discussed in Sect. 4.1) are evaluated, but only the best ones are shown.
2. Conjugate gradient iteration: the spMV product is the most time-consuming part of the conjugate gradient method and the only one that depends on the matrix layout. The performance of different storage and calculation schemes was analysed.
3. Data conflicts: race conditions can be handled in two ways: with the use of atomic operations or colouring. The former is straightforward and does not require any special implementation considerations. Optimal colouring on the other hand is an *NP-hard* problem, but suboptimal colouring algorithms are fast. The GPU algorithm uses two levels of colouring: thread and block colouring. Blocks with different colours are executed after each other by a kernel call. Memory writes for threads

with different colours are separated by a *__syncthreads*() call. To decrease the number of synchronisations, these writes were buffered and grouped.

4. CPU and GPU: the algorithms are implemented as CPU codes using OpenMP threads executing Algorithm 2 by blocks of elements. In the GPU version one element is assigned to each thread.
5. Renumbering to improve cache efficiency: several cache blocking renumbering schemes have been applied to improve cache locality.
6. Preconditioning: a simple Jacobi or an SSOR precoditioner are included in the conjugate gradient iteration, and their impact on performance is analysed.

5.4 CPU Implementation

Comparing single-core CPU performance to the performance of GPUs with hundreds of processing cores is not realistic since modern systems have multiple CPU cores. To provide a fair comparison, we implemented the assembly and the solution phase using OpenMP. Our test system included a 2 socket Intel Xeon X5650 processor, with a total of 12 physical cores, 24 with hyper-threading enabled.

The CPU implementations use a block-colouring approach to avoid race conditions, each thread works on a block of quadrilaterals and no two blocks with the same colour have neighboring quadrilaterals. During assembly in these implementations we do not do any redundant computations, jacobians and local local quadrature point coordinates are precomputed and stored in local arrays because these fit easily in the cache of the CPU. To further optimise performance, threads were pinned to CPU cores through the *KMP_AFFINITY* = *compact* environment variable to avoid migration between sockets. To minimise the effects of non-uniform memory access (NUMA), memory is initialised from within OpenMP loops corresponding to further computational loops so as to make sure that memory most frequently used by different threads during the assembly and the solution process is allocated to physical memory attached to the appropriate CPU socket (this relies on the first touch page allocation policy of the operating system). Every CPU figure below shows performance using 24 threads.

The benefit from having more threads is clear in both phases, using 24 threads there is a 7–12 times speedup over a single thread during assembly, and a speedup between 5 and 8 during solve.

6 The CSR Layout

The compressed sparse row format (CSR) is one of the most widely used sparse storage formats, supported by several libraries such as NVIDIA's CUSPARSE [19], which implements several kinds of sparse matrix operations. In the case of the finite element method the matrix layout can be calculated from the mapping M_e and used in the assembly phase to find the location of a nonzero within its row. Our tests use this approach but it is out of scope of this paper to optimize this precomputing phase; in practice we do this on the CPU and use it as an input to the algorithm.

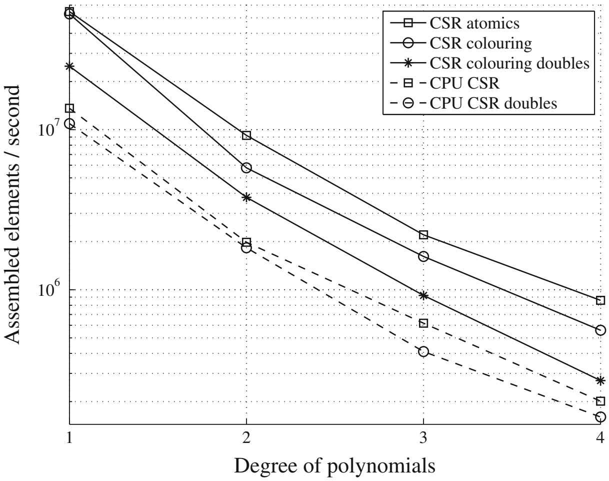


Fig. 5 Number of elements assembled and written to the global stiffness matrix using the CSR storage format

6.1 Assembly Phase

In the assembly phase described by Algorithm 2 the global memory address of K_{ij} has to be determined by accessing the row and column pointers for the current local matrix element, then writing the value while avoiding write conflicts either via atomic operations or colouring. Only the column pointer lookup can be coalesced due to the unstructured nature of the grid. The impact of this overhead decreases with the increasing computational requirements of higher order elements. Figure 5 shows a speedup of up to 4.5 times over the CPU.

6.2 spMV Phase

There are two approaches to evaluate the matrix-vector product: assign one thread per row and add up the products with the multiplicand or to assign multiple threads per row thereby achieving coalesced memory access to the values in the same row. The partial sums in the latter case are added up by a binary-tree reduction algorithm. We investigate the effects of assigning different number of threads to each row of the sparse matrix. The number of threads per row is directly related to coalesced memory accesses, but with the introduction of implicit caches in the Fermi architecture the performance is also affected by cache hits and misses. In Fermi, each global memory access loads a whole cache line (128 bytes) and since the size of L1 cache is at most 48 kBytes only a very limited number of them can be actually reused. This results in a high amount of cache misses. As shown in Fig. 6 the optimal number of threads assigned to each row varies with the degree of elements, which is directly related to the length of the rows. The difference between the best and worst choice can be as much as 10:1.

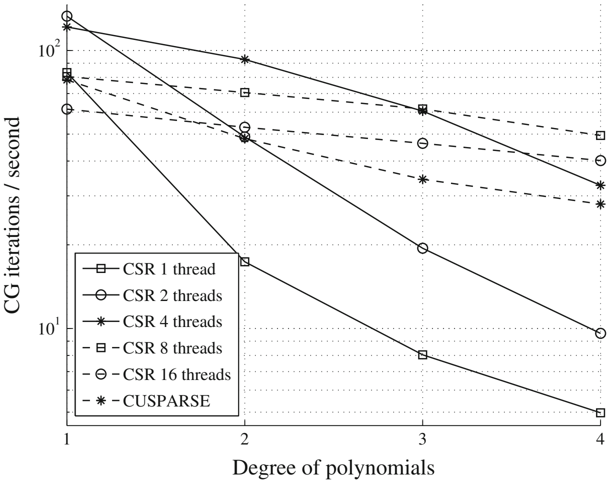


Fig. 6 Number of CG iterations per second on a 4 million row matrix using the CSR storage format. During the spMV, different numbers of GPU threads were assigned to each row of the sparse matrix

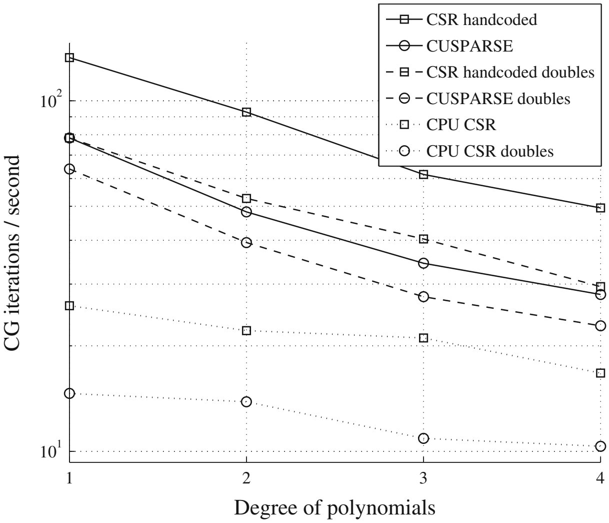


Fig. 7 Number of CG iterations per second on a 4 million row matrix using the CSR storage format. The spMV was performed by a hand-coded kernel and by CUSPARSE

Figure 7 shows the performance of only the optimal versions of our CSR spMV kernel compared to CUSPARSE and CPU versions. By always evaluating and assigning the optimal number of threads to process the rows of the matrix, our algorithm, compared to CUSPARSE, shows a speedup of up to 2 times in single and 1.4 in double precision and between 3 to 5 times over the CPU. A heuristic decision algorithm for the number of threads assigned to process each row was presented in [25] and submitted to NVIDIA, it will be part of a future release of CUSPARSE.

7 The ELLPACK Layout

The ELLPACK storage format has gained popularity for use on the GPUs because of its aligned rows as shown in Fig. 3. The population of the matrix can be done in a similar way to CSR—precomputing the matrix layout and writing to those memory locations in a thread-safe way. The other possibility is to allocate an empty matrix with an extra field for each row that stores the current length of that row. The maximum length of the rows can be upper-bounded by the number of degrees of freedom in each element and the maximum degree of vertices in the mesh. Then in the assembly phase each new value and its column index are appended to the end of the row. This of course requires more memory space allocated for the matrix, but when computing the local matrices row by row, each row can be written to global memory safely by increasing the row length only once with the number of new values to be written. After the assembly, these rows can be consolidated by sorting them by column indices and adding up values with the same index. This approach will result in the same matrix as if the layout was precomputed. The other approach we call 'lazy ELLPACK' is to leave the rows as they are, which will still produce a valid result in the spMV phase, but the rows will be longer. The relative number of multiple entries for the same column index in a row decreases as the order of elements increases, making this overhead smaller.

7.1 Assembly Phase

Computationally the assembly phase is exactly like the others, only the memory access patterns differ. In the lazy scheme less memory input is required as only one pointer (the row length) per local matrix row is necessary to determine where to write its values in global memory, unlike the precomputed version, where a pointer is required for each local matrix element—but this information can be laid out in memory so the access to it will be coalesced. When writing data to memory the lazy scheme has to store column indices for every value, in the precomputed version this is already known. Figure 8 shows that the two schemes perform almost the same when using atomics to avoid race conditions, but when using colouring the lazy version outperforms the precomputed one as it requires less synchronisation. The atomic and lazy approaches have a speedup of up to 20 times over the CPU in single precision and up to 3 times in double precision mode.

7.2 spMV Phase

In our tests, one thread is assigned to each row of the matrix, and since the matrix is transposed in GPU memory, these threads are reading the values and column indices of the matrix in a coalesced way—that is if the rows had the same length. Due to caching in Fermi GPUs, even then whole cache lines will be loaded resulting in excess use of bandwidth. The lazy scheme has similar issues, but the imbalance between the length of the rows is worse—when a degree of freedom belongs to more elements, its row in the stiffness matrix will have more values with the same column index. As shown in Fig. 9, the GPU is up to 6 times faster than the CPU in single and 5 times in double precision.

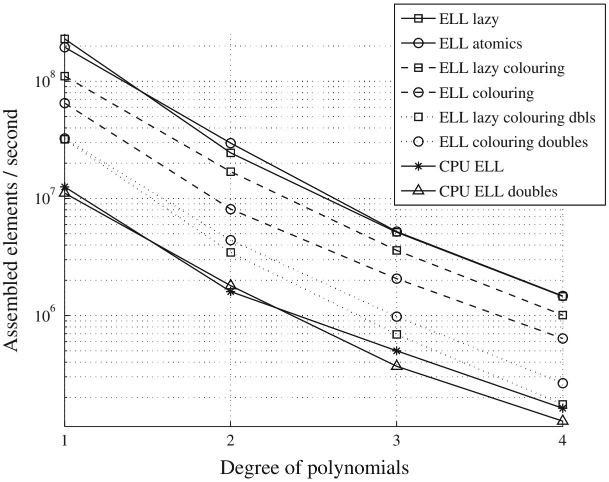


Fig. 8 Number of elements assembled and written to the global stiffness matrix using the ELLPACK storage format. The lazy version does not write to predetermined memory locations, but appends every new value to the end of the row

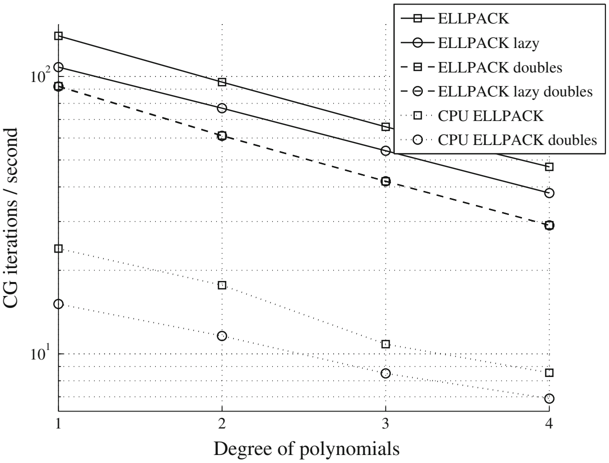


Fig. 9 Number of CG iterations per second on a 4 million row matrix using the ELLPACK storage format. The rows of the lazy version are not sorted and may have multiple values for the same column index

8 The LMA Method

The local matrix assembly method is based on the fact that in the iterative solver, the stiffness matrix K is not required explicitly. This approach circumvents the sparse matrix issues that arise when dealing with unstructured grids by storing stiffness values on a local matrix basis. This enables completely coalesced access to stiffness values in both the assembly and the spMV phase—at the expense of redundant storage. Contibuting stiffness values related to degrees of freedom that are on edges or vertices

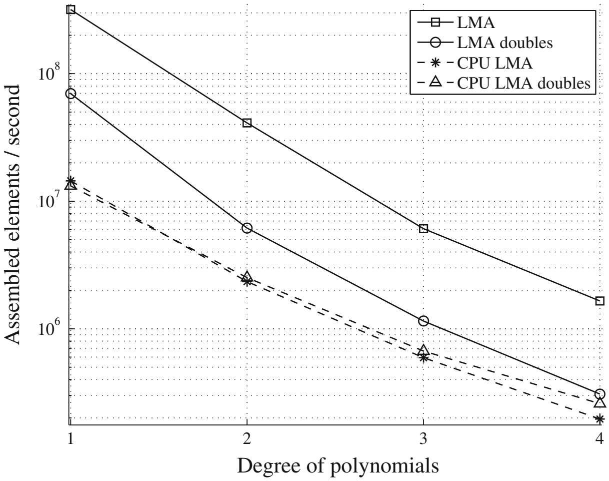


Fig. 10 Number of elements assembled and written to the global memory using the LMA method

connecting elements are stored in the local matrices of those elements. This overhead decreases as the order of elements increases.

8.1 Assembly Phase

The assembly phase requires the mapping from elements to node indices and the node coordinates to be read from global memory—the only part of the LMA method that cannot be coalesced, all writes to global memory are aligned by the thread index. Also an important aspect of LMA assembly is that there are no write conflicts as each thread writes to its own memory space. As shown in Fig. 10, the speedup of the GPU over the CPU is up to 20 times in single and 5 times in double precision.

8.2 spMV Phase

The spMV kernel is slightly different from the others as it is based on elements and not the rows of the matrix. Given these small dense local matrices, it is possible to efficiently exploit the symmetric nature of the stiffness matrix. In the implementation of the multiplication this would mean a nested loop with the bounds of the inner loop depending on the outer loop. However, the compiler is not able to create an efficient machine code from such a nested loop, so we unrolled the whole dense matrix-vector multiplication by hand. While CSR and ELLPACK spMV multiplications have to read a column index for each value in the stiffness matrix, the LMA method uses the mapping M_e to get row and column indices. Thus, as we have shown in Sect. 4.3, at higher degrees the data moved by LMA is significantly less then the data moved by global matrix approaches.

In this scenario however, more threads can increment the same value of the product vector; thread safety is guaranteed by the use of atomics or colouring. Figure 11

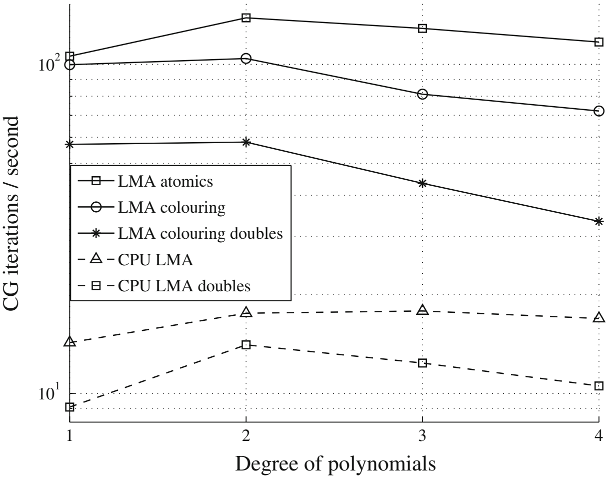


Fig. 11 Number of CG iterations per second on a grid with 4 million degrees of freedom using the LMA method

clearly shows the penalty incurred by the synchronisation between colours. As the spMV phase is bandwidth limited, and those memory accesses which may conflict (i.e. writing to the product vector) are probably not coalesced anyway, a separate memory transaction has to be issued for each atomic operation and each coloured write alike. This makes the overhead of synchronisation between colours a key limiting factor. The GPU achieves speedups of up to 8 times in single and 4 times in double precision over the CPU.

9 The Matrix-Free Method

The matrix free method is very similar to the LMA method, but instead of writing the local matrices to global memory, it uses them to perform the matrix-vector product, thereby fusing the assembly and the spMV phase. This of course means that the local matrices have to be recalculated every time the spMV multiplication is performed. The matrix-free approach has the advantage of not having to move the stiffness matrix to and from memory, which saves a lot on bandwidth. On the other hand calculating local stiffness matrices is increasingly more computationally expensive with higher order elements. As a result, the matrix-free approach can only be better than the other methods, if the computation of the local matrices is faster than moving them from global memory to the chip. Figure 12 shows a clearly steeper decline in performance compared to the spMV phase of other methods, which means that at higher degree elements the matrix-free method is more compute limited than the other methods are bandwidth limited. On current hardware, the gap between memory transactions and computations is not wide enough to support such a high amount of redundant computations.

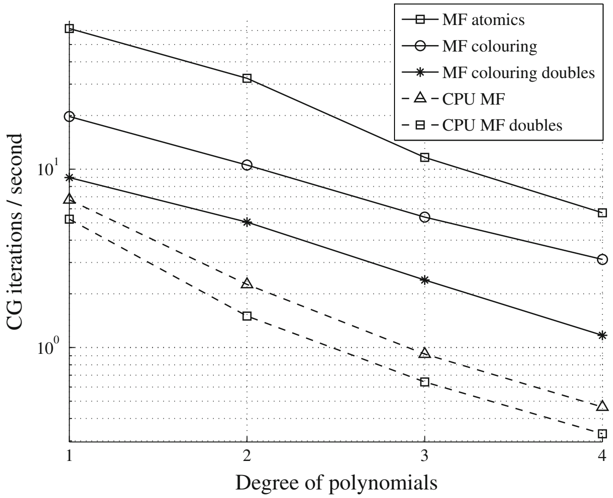


Fig. 12 Number of CG iterations per second on a grid with 4 million degrees of freedom using the matrix-free method

10 Bottleneck Analysis

When trying to optimize algorithms to run faster on the GPU, the most important thing to investigate is whether thay are compute or bandwidth limited. The NVIDIA Tesla C2070 has a theoretical maximum bandwidth of 144 GB/s, and a compute capacity of 1,03 TFLOPS for single and 515 GFLOPS for double precision calculations [21], although these numbers assume fully coalesced memory accesses and purely fused multiply-add floating point operations, which have a throughput of one instruction per clock cycle, but are counted as two operations. Applications rarely achive more than two thirds of the theoretical peak performance. Unstructured grids deal with scattered data access and a high amount of pointer arithmetics, both of which make the utilization of GPU resources difficult. The introduction of L1 and L2 implicit caches with Fermi helps achieve better memory bandwidth, but in some cases it can degrade performance: cache trashing can result in a high fraction of the moved data being unused. As a comparison, the Intel Xeon X5650 system has a theoretical performance of around 120 GFLOPS when SSE vectorisation is fully utilised, which amounts to 10 GFLOPS per core.

On the computational side, there are certain operations that are a natural fit for the GPU and provide high throughput, such as floating point multiply and add (32 operations per clock cycle per multiprocessor). Some integer operations are more expensive: 32 operations per cycle for addition but only 16 for multiplication and compare. The most expensive operations are floating point division and special functions such as square root, sine, cosine, and exp, which have a throughput of 4 operations per cycle per multiprocessor.

From a single thread perspective the algorithm is a sequence of memory operations and compute operations. To utilise the maximum amount of bandwidth there have to be enough threads and enough compute operations so that while some threads are waiting for data from memory, others can execute compute instructions. In theory, for the Tesla C2070 to operate at maximum efficiency from both the memory and compute perspective there have to be 28 floating point multiply-add operations for every single precision floating number loaded from global memory. In practice because of control overhead, other kinds of operations, and caching this number is significantly less (around 10).

The NVIDIA Visual Profiler gives very useful hints to decide whether a GPU kernel is compute or memory limited, but it also displays metrics such as ratio of divergence, cache statistics etc. which can be very helpful when analysing non-trivial aspects of an algorithm.

10.1 The Assembly Phase

As the GPU has significantly less resources per thread than the CPU, it is improtant to investigate the balance of computations and communications when it is possible to trade one for the other. As described in Sect. 4.1, we have implemented three different approaches, the first using local memory to precompute the coordinates of local quadrature points to save on computations in the innermost loop (labeled as *Local reuse*), the second which recomputes these coordinates and hence it is the most computationally intensive (labeled as *Redundant compute*) and the third that interchanges the loop over quadrature points with the loop over pairs of degrees of freedom, performing the least amount of computations but resulting in increased memory traffic because the stiffness values, residing in global memory, are updated repeatedly (labeled as *Global reuse*). Figure 13 shows that the third approach on the GPU is not a viable option, however for low degree polynomials the L1 cache can contain the register spillage resulting from the increased memory footprint of the first approach making it faster than the redundant compute version. For higher degree polynomials this is no longer

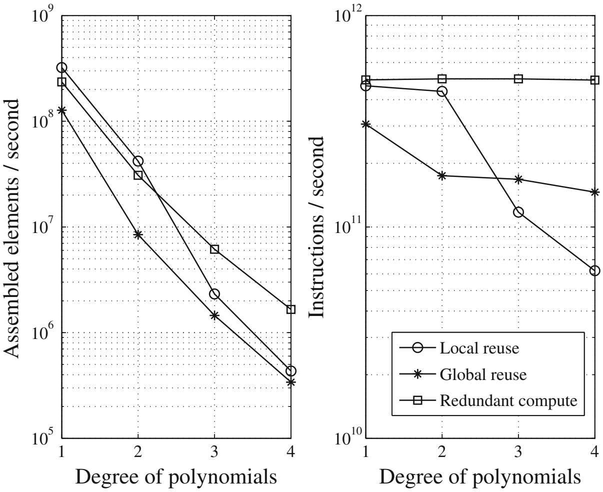


Fig. 13 Number of elements assembled and instruction throughput using assembly strategies that trade off computations for communications. Values are stored in the LMA format

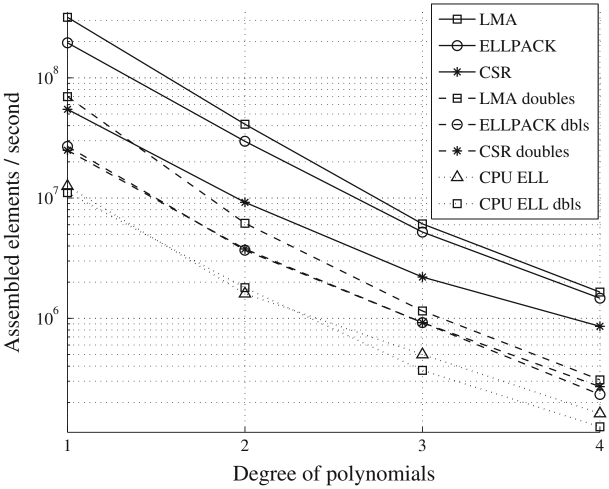


Fig. 14 Number of elements assembled and written to global memory when using different storage formats and storage precision

true, the precomputed values are spilled to global memory resulting in a dramatic drop in instruction throughput and performance. The second approach scales very well with the increasing degree of polynomials, using the same amount of registers and showing a steady instruction throughput rate.

Both the assembly and the spMV phases have to move the entire stiffness matrix to or from memory, and this transfer makes up the bulk of all memory traffic in both phases. Looking at the performance degradation in Fig. 14, as the order of elements increase, it is apparent that the assembly becomes much slower than the spMV. This means that while increasing the order of the elements results in having to move more data to and from memory, it also requires more computation in the assembly phase because the number of quadrature points also increases as a square function (in 2D) of the degree of polynomials used. Figure 15 shows a slight increase in instruction throughput for the LMA and ELLPACK approaches, and Fig. 16 shows quickly decreasing bandwidth in the assembly phase. These factors indicate that the assembly phase is compute limited. CSR throughput figures on the other hand show an increasing tendency, while its bandwidth utilisation is the same as that of the other two approaches. The reason for this is the high percentage of cache misses; while threads writing to LMA and ELLPACK data layouts work on a small set of cache lines at the same time, thanks to their transposed data layout, threads writing to the CSR layout do not.

Based on these observations, it can be stated that the assembly kernel is increasingly compute limited with higher order elements. According to the Visual Profiler's output and our own calculations the instruction throughput of the LMA approach is around $500 * 10^9$ instructions per second, which is a good proportion of theoretical 1TFLOPS throughput considering the amount of integer arithmetic and control overhead—as a comparison, the CUBLAS [22] dense matrix-matrix multiplication benchmark reports 630 GFLOPS in single and 303 GFLOPS in double precision. It is also interesting to see the performance penalty for colouring (2 to 4 times)—that is the cost of synchronisation within thread blocks and the multiple kernel calls for different block colours.

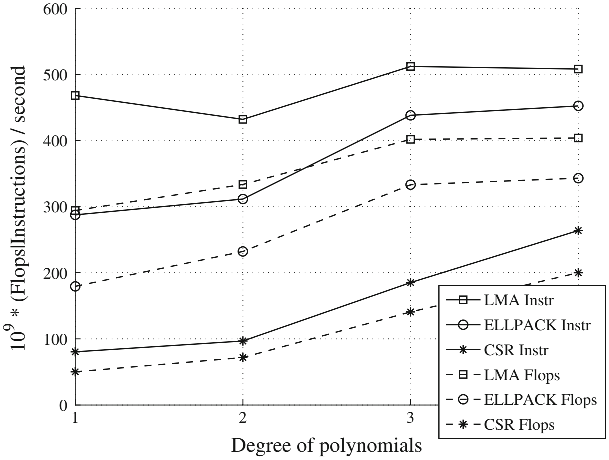


Fig. 15 Number of general and floating point instructions executed in the assembly phase by different approaches on a grid with 4 million degrees of freedom

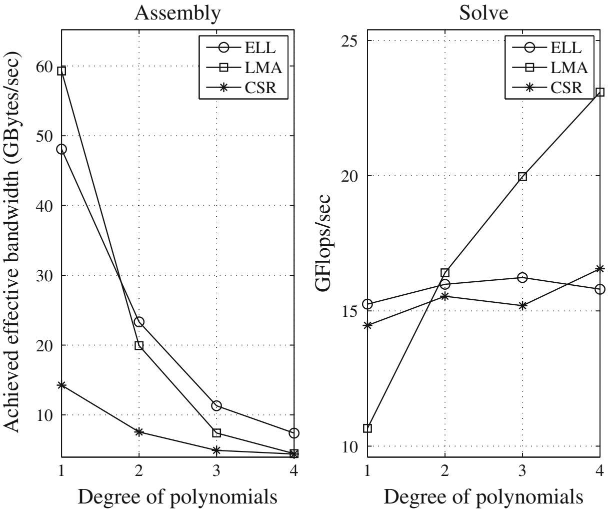


Fig. 16 Achieved effective bandwidth in the assembly phase and number of floating point instructions in the spMV phase by different approaches on a grid with 4 million degrees of freedom

When moving to double precision, it can be seen in Fig. 14 that the LMA assembly performance performance is only a fourth of its single precision version. Double precision ELLPACK and CSR have to use colouring due to the lack of native support for atomic operations with doubles. In the case of the CPU, the compute limits are again apparent, even though the CPU versions have to perform less computations at the expense of having to store local quadrature points and their jacobians. Regardless of the memory access pattern, all versions perform similarly.

Using LMA, the GPU's speedup over the CPU is between a factor of 10 and 30 in single precision using atomics, and between 2.5 and 7 in double precision. The speed difference in the actual calculations and in the theoretical performance of the GPU versus the CPU are very close.

10.2 The spMV Phase

The sparse matrix-vector product is commonly known to be a bandwidth limited operation [3]. In fact it has to move just a little less data than the assembly phase, but the number of operations is significantly less: approximately one fused multiply-add for each nonzero element in the matrix. Figure 16 clearly shows a low instruction throughput compared to the theoretical maximum. Using the LMA approach incurs an overhead of having to avoid race conditions using atomics or colouring. The LMA approach offers fully coalesced access to the elements of the stiffness matrix, and both ELLPACK and CSR use optimizations to improve bandwidth efficiency. However, the access to the elements of the multiplicand vector is not coalesced, though caching can improve performance. Global matrix approaches using CSR store the least amount of stiffness data, but they have to read a column index for every nonzero of the sparse matrix. The LMA approach uses the mapping M_e to get row and column indices, as shown in Sect. 4.3 the local matrix approach has to move significantly less data, especially at higher degree polynomials. Figure 17 shows that global matrix approaches have very similar performance, but the less data-intensive local matrix approach provides the best performance for higher order elements. As expected, the difference increases with the increasing degree of elements. This also supports the conclusion that the sparse matrix-vector product is bandwidth limited. Furthermore Fig. 18 shows the bandwidth utilisation of different approaches. The ELLPACK layout shows the best bandwidth utilisation, but since it has to move more data it is still up to 50 % slower than the LMA layout. Although using either the CSR or the ELLPACK layout results in having to move the same amount of useful data, the transposed layout of ELLPACK provides

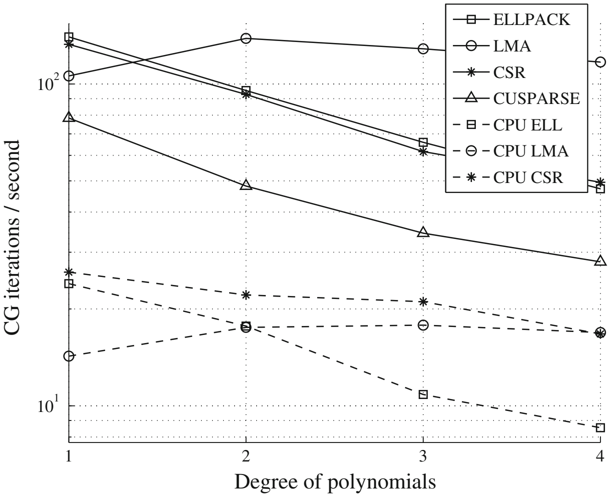


Fig. 17 Number of CG iterations per second with different storage formats at single precision

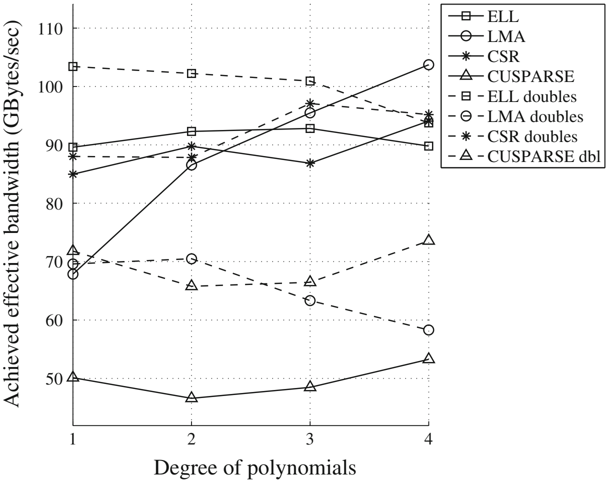


Fig. 18 Achieved effective bandwidth in the spMV phase by different approaches on a grid with 4 million degrees of freedom

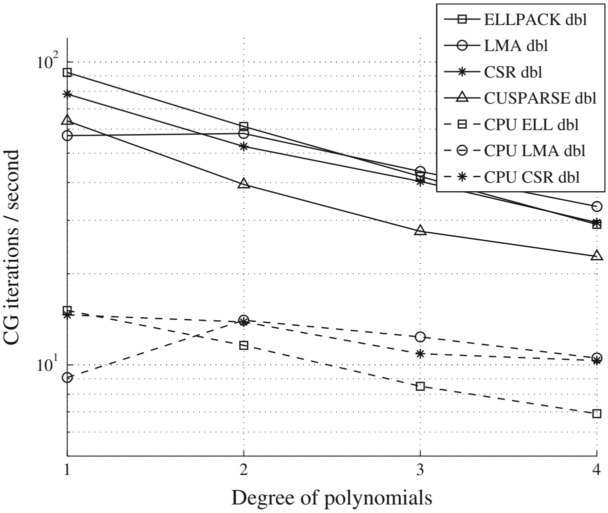


Fig. 19 Number of CG iterations per second with different storage formats at double precision

up to 10 % higher effective bandwidth. The zeros padding the rows of ELLPACK are not factored into these figures. Figure 19 shows that in double precision, the performance of the LMA approach falls back because colouring has to be used to avoid race conditions.

The GPU's speedup over the CPU using global matrix approaches is up to 5 in single precision and 3 in double precision. Local matrix approaches outperform the CPU by up to 7 times in single and 5 in double precision.

10.3 Occupancy and Block Size

For the GPU to achieve maximum instruction throughput there have to be enough threads to hide the latency of access to global memory and to have a full instruction pipeline: in GPU terms this is called occupancy and it is measured as the fraction of active threads per multiprocessor versus the theoretical maximum, which on Fermi is 1536. The most important factors determining occupancy are the number of registers each thread uses and the amount of shared memory used by each block. Also an important factor that is tied to the first two is the size of the blocks: each multiprocessor can have up to 8 active blocks at the same time. The number of registers and shared memory available per multiprocessor is limited: 32768 and 48 kBytes respectively. Thus using a high amount of these resources may limit the number of active blocks. Furthermore, when threads within the block have to synchronise—e.g. when using colouring in the assembly phase—the more threads there are in a block the more overhead it poses. It is important to find the best block size to achieve the best performance. Since the spMV phase is bandwidth limited and the usage of shared memory and registers is very low there are no occupancy or synchronisation problems, so this section focuses on finding the best blocksize for the assembly phase.

Tests are run using an autotuning framework called Flamingo [26]. Performance is evaluated at several block sizes and the degree of elements ranges from one to four.

Figure 20 shows the performance of assembly with different order elements at different lock sizes. The assembly phase of the LMA approach has no write conflicts thus no need for synchronisation, nor does it use any shared memory, so the occupancy varies based on register usage and size of blocks. For these reasons—as the figure shows—there is a small variation is the performance when using different block sizes: for low order elements below 2 %, for higher order elements up to 10 %.

The ELLPACK storage format requires thread safe access to the global matrix via either atomics or colouring. The colouring approach has to ensure this by using

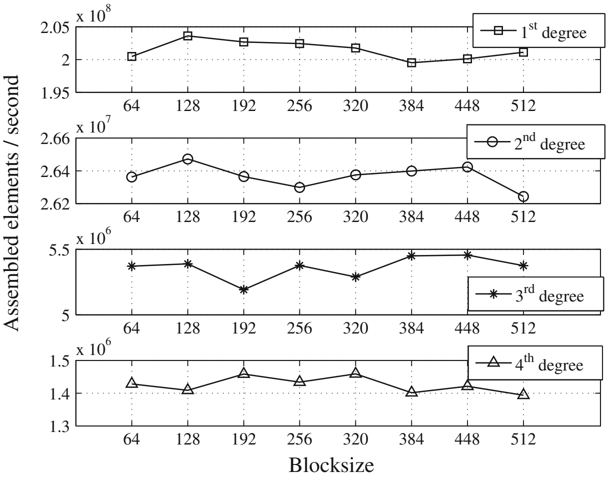


Fig. 20 Number of elements assembled at different block sizes using the LMA approach

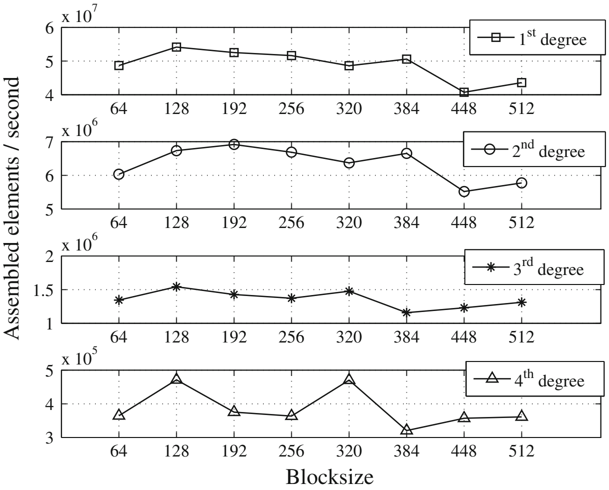


Fig. 21 Number of elements assembled at different block sizes using the ELLPACK approach with colouring

synchronisation between different colours, which as shown in the previous section has a significant overhead. In this case the number of threads within a block factors heavily into the cost of synchronisation. Figure 21 shows variations around 20 % and at high order elements the difference can reach even 50 %.

10.4 Renumbering and Cache Blocking

As it was shown earlier, the assembly phase of the finite element method is compute bound, thus it cannot benefit significantly from better caching. The sparse matrix-vector multiply on the other hand is heavily bandwidth limited. Access to the multiplicand vector is not coalesced, so accessing it element-by-element wastes bandwidth—this is where Fermi's L1 and L2 cache can help. The size of these caches is relatively small compared to the number of threads running concurrently, so by using the well-known cache blocking method in combination with the renumbering of the degrees of freedom, better caching efficiency can be achieved. Renumbering can also improve the efficiency of the ELLPACK spMV by reordering degrees of freedom that have similar row length to be adjacent to each other in the global stiffness matrix as shown in Fig. 22. This way when reading the matrix there will be minimal divergence between the threads iterating through those rows. In this section we show the effects of these methods on the ELLPACK storage format, which can potentially benefit from both blocking and renumbering. Figure 23 shows the effects of different combinations of these methods: "Original" uses the same numbering scheme as in the previous tests, "blocked renumbering" uses the same method of numbering as before, but blocks the iteration through elements, "aligned renumbering" uses the scheme this section describes, and "blocked aligned renumbering" combines blocking and aligned renumbering. 16k and 48k designate the size of the L1 cache used.

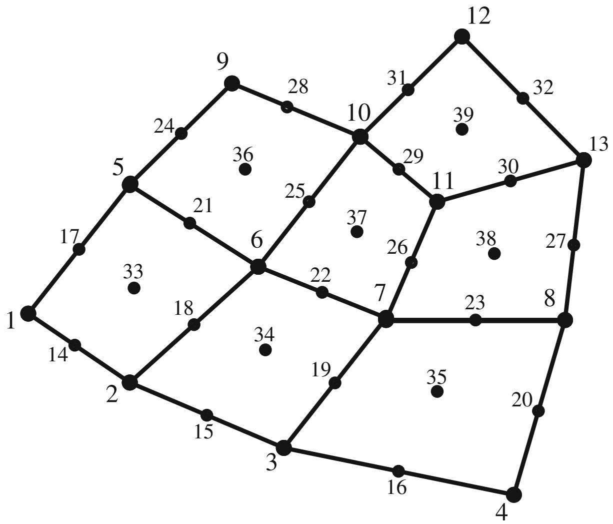


Fig. 22 Renumbering the degrees of freedom in a mesh in order to have aligned length rows in the stiffness matrix

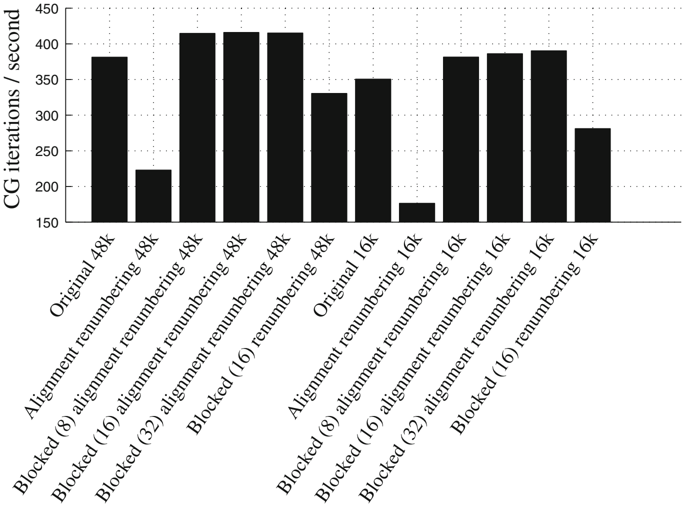


Fig. 23 The effect or renumbering and blocking schemes on a 1 million row ELLPACK matrix

1. *Renumbering for aligned length rows:* This renumbering scheme is simple: iterate through every vertex of the mesh, then every node that is on an edge, and at last the degrees of freedom that are inside the elements. Figure 22 shows an example of this reordering. This will balance the length of the adjacent rows, because unless vertices have highly varying degrees or many constrained neighbours, they will have a non-zero integral with the same number of adjacent basis functions, and thus have the same row length. As shown in Fig. 23, this scheme performs poorly despite almost 0 % divergence. The reason for this is the high number of L1 cache

misses and thus the high number of instructions reissued: about three times as more for 16 kB L1 cache than in the original numbering scheme. Increasing the size of the L1 cache to 48 kB decreases the number of cache misses, but the performance is still poor.

2. *Blocking:* The blocking scheme partitions the grid into blocks of $n*n$ geometrically close elements and renumbers their degrees of freedom. This serves to improve cache locality, as threads will access elements of the multiplicand vector that are close to each other. As shown in Fig. 23 blocking alone did not improve performance either.
3. *Blocking and aligned renumbering:* Combining both approaches results in a numbering scheme, where vertices of geometrically close elements have sequentially ordered indices, so do nodes on edges and inside elements. This gives a performance improvement of about 10 %.

The CPU version behaves similarly, the combined blocking and aligned renumbering approach gives a 10 % performance improvement, while other methods have slightly worse performance compared to the original numbering scheme.

11 Preconditioning

When solving ill-conditioned problems with a high condition number, one usually relies on preconditioners to ensure and accelerate convergence during the iterative solution phase. This consists of inserting a step in the iterative algorithm that performs the preconditioning:

$$z = M^{-1}r, \tag{22}$$

where M is the preconditioning matrix. These preconditioners can range from simple diagonal (Jacobi), popular for its ease of use, through SSOR, Cholesky and LU factorisation, to the most powerful (and most expensive) multigrid-based preconditioners [2]. Here we evaluate the performance implications of applying the Jacobi precoditioner, defined as follows:

$$M_{ij} = \begin{cases} K_{ij} & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases} \tag{23}$$

where K is the stiffness matrix, and the Symmetric Successive Over-Relaxation (SSOR), defined as:

$$M = (L + D)D^{-1}(L + D)^T, \tag{24}$$

where L is the strictly lower triangular part of K and D is the diagonal. The Jacobi preconditioner is trivially parallel, however the SSOR enforces an ordering during the solution of the lower and the upper triangular systems, thus we use a red-black variant in conjunction with a full graph colouring algorithm that ensures that no two nodes that are connected via an edge are updated at the same time and provides an

Algorithm 3 Coloured SSOR preconditioning.

```
Compute $z$ defined by:
$(L + D)D^{-1}(L + D)^T z = r$
Sub-step: solve $(L + D)y = r$ for y:
for colour $c = 1 \ldots ncolors$ do
  for each row $i$ with colour $c$ in parallel do
    $y_i = (1/D_i)(r_i - \sum_{columns\ j\ with\ colour<c} K(i, j)y_j)$
  end for
end for
Sub-step: solve $D^{-1}(L + D)^T z = y$ for z:
for colour $c = ncolors \ldots 1$ do
  for each row $i$ with colour $c$ in parallel do
    $z_i = y_i - (1/D_i) \sum_{columns\ j\ with\ colour>c} K(i, j)z_j)$
  end for
end for
```

ordering for the upper and lower triangular solves [1]. The algorithm is described by Algorithm 3:

The diagonal preconditioning adds very little overhead to the conjugate gradient iteration; once the diagonal values are extracted from the stiffness matrix, it is a matter of a simple elementwise vector-vector multiplication, and so it has virtually no impact (less than 10 %) on performance when compared to performance shown in Figs. 17 and 19. The coloured SSOR on the other hand involves several kernel launches, for each colour, in both the forward and the backward solution steps. In theory, the amount of useful computations carried out during preconditioning is almost the same as during a single sparse-matrix vector product. Performance figures are shown in Figs. 24 and 25. It can be observed that even for global matrix assembly approaches, the performance penalty is much higher than two times, which is due to the coloured execution: no two

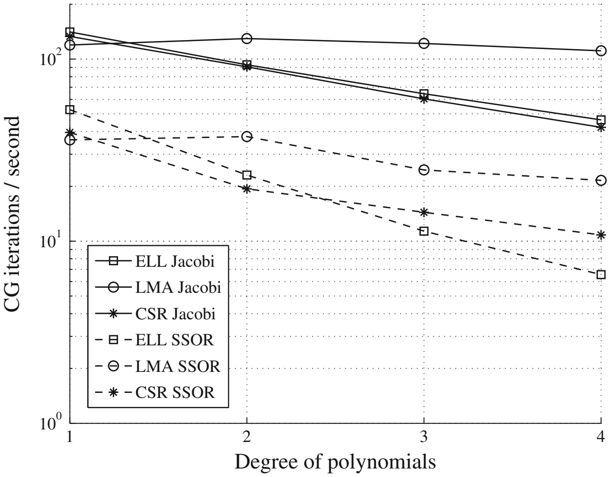


Fig. 24 Number of CG iterations per second with different preconditioners and storage formats in single precision

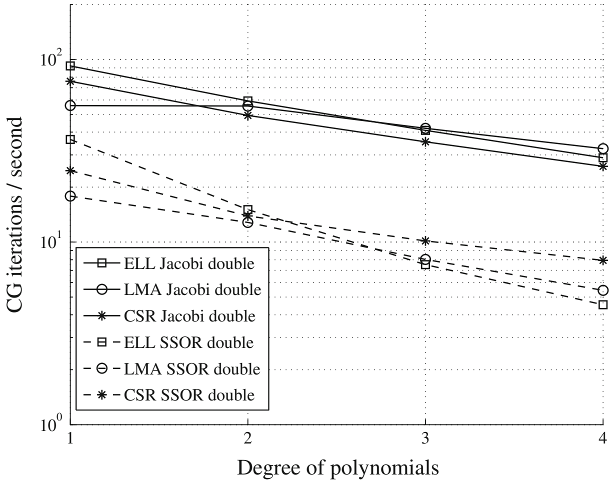


Fig. 25 Number of CG iterations per second with different preconditioners and storage formats in double precision

degrees of freedom in the same element can have the same colour, resutling in at least $(degree + 1)^4$ colours.

The CSR format is least affected by the coloured execution, because multiple threads are assigned to compute the product between any given row of the matrix and the multiplicand vector, which ensures some level of coalescing in memory accesses. ELLPACK on the other hand stores data in a transposed layout that gives good memory access patterns during an ordinary spMV, however during coloured execution neighboring rows do not have the same colour thus neighboring threads processing rows of the same colour are accessing rows that are far apart, resulting in very poor memory access patterns due to big strides and low cache line utilisation. Because of these issues, the performance of the ELLPACK layout on the SSOR falls rapidly with the increasing degree of polinomials used. When executing SSOR using the LMA layout, threads are still processing neighboring elements, however there is no opportunity of data reuse within an element because by definition all its degrees of freedom have a different colour. Still, LMA's data layout enables more efficient memory access patterns, granting it an edge over global assembly methods in single precision, but the requirement for a two-level colouring approach in double precision in order to avoid race conditions between neighboring elements reduces its performance below that of CSR.

Even though ELLPACK is very well suited for simple operations such as the spMV, it does not scale well to larger degree polynomials. LMA performs well on simple operations and single precision, but the lack of support for atomics in double precision and the complexity of the implementation for preconditioners like SSOR do not make it the obvious best choice. Furthermore LMA restricts the choice of preconditioners: for example an incomplete LU factorisation with fill-in cannot be applied, because the layout can not accommodate the additional nonzeros. However, a geometric or p-Multigrid scheme [10] could be a good choice in combination with a Jacobi smother;

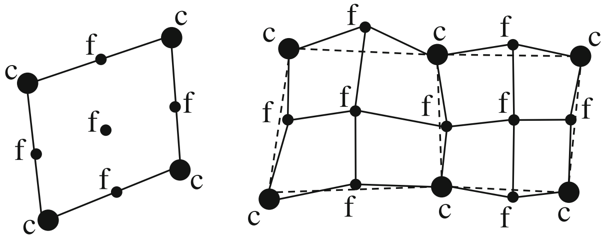


Fig. 26 Two examples of multigrid restrictions that maintain the element structure: p-multigrid, going from degree 2 polynomials to degree 1, and a geometric multigrid variant. *Nodes* marked c exist on both the coarse and fine grid levels, and *nodes* marked f only exist on the fine grid level

the LMA format has the geometric information readily available and an element structure can be maintained on the coarser levels with restriction schemes such as shown in Fig. 26. The implementation of such a multigrid preconditioner is out of scope of this paper and will be addressed by future research.

12 Conclusions

A comprehensive study of the finite element method has been presented that investigates the performance of finite element assembly and solution on the GPU. The main contributing factors are found to be the balance of computations and communications, and the layout and access pattern of memory storing the stiffness values. We present three approaches to the assembly algorithm that trade off computations for communications:

1. redundant computations,
2. increased thread-local private memory,
3. increased global memory traffic.

We show that on the GPU the very limited amount of resources per thread and the ratio of bandwidth to off-chip memory and computational throughput makes the most computationally intensive alternative often the best choice. Because of very limited cache size, the third approach becomes memory bandwidth limited and performs the worst in all cases. Similarly, because of the limited number of registers and cache size, the increasing local memory requirements of the second approach become a limiting factor at higher degree polynomials making it 30 % faster for low degree polynomials but up to four times slower for higher degree ones. The most computationally intensive approach has the additional benefit of scaling well to any degree of polynomials, showing a steady throughput rate.

We have also analysed two main approaches to the storage and use of stiffness values: the local matrix approach and the global matrix approach using either the CSR or the ELLPACK storage format. The implications of the choice of storage approach are analysed in detail: performance bottlenecks resulting from memory access patterns, amount of computations, handling of race conditions, use of resources and occupancy. The local matrix assembly is demonstrated to be a viable alternative to global assembly, providing the best performance in most cases at the expense of more programming effort, thanks to the reduced data transfer requirements during the sparse matrix-vector multiplication phase. A variation of the local matrix approach, the matrix-free method, which avoids the storage of stiffness data completely by recomputing them on-the-fly during the iterative solution is shown to become heavily compute-limited at higher order elements.

The most widely used sparse matrix storage format CSR (compressed sparse row) proves to be a slightly worse choice than LMA or ELLPACK, being on average two times slower in the assembly phase due to suboptimal access patterns. The number of assembled quadrilateral elements per second outperforms triangular assembly shown

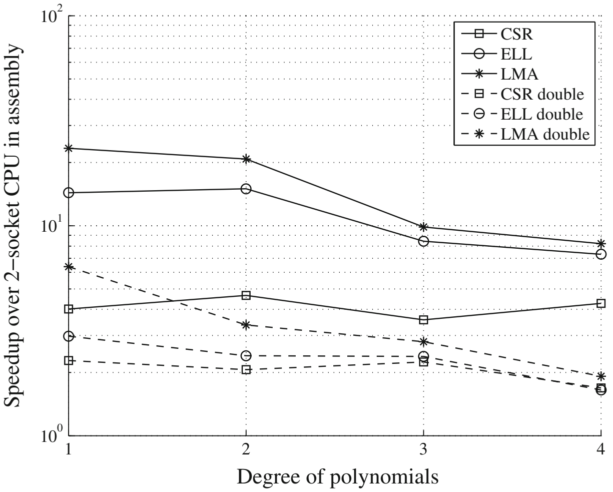


Fig. 27 Speedup of the GPU in the assembly phase over a 12-core CPU using the CSR format

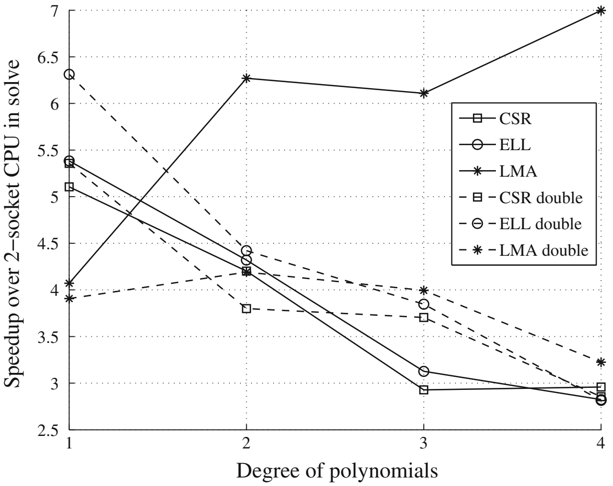


Fig. 28 Speedup of the GPU in the solution phase over a 12-core CPU using the CSR format

in [6,18] by a factor of up to 10. The storage schemes are also compared during the iterative solution phase; when using no or diagonal preconditioning ELLPACK and LMA still perform the best, however with a more complicated SSOR preconditioner we show ELLPACK's inability to scale and the advantage of the CSR layout in double precision. LMA somewhat restricts the choice of preconditioners; while it gives good performance on SSOR and could efficiently support certain multigrid schemes with a Jacobi preconditioner, it would not be possible to implement more complicated algorithms, such as ILU. NVIDIA's CUSPARSE library is also evaluated in the spMV phase, its performance being on average 20–50 % lower than our hand-coded kernels.

Speedups of up to 20 times in the assembly phase and 7 in the solution phase are shown in Figs. 27 and 28 for single precision (6 and 5 respectively in double precision), over a fully utilised 12-core Xeon CPU. Furthermore, several GPU-specific factors are analysed such as autotuning for optimal occupancy and renumbering schemes for better caching.

Acknowledgments This research was supported in part by the UK Engineering and Physical Sciences Research Council through project EP/J010553/1 on "Algorithms and Software for Emerging Architectures", and in part by the EU LLP/Erasmus program 10/2010-2011/Erasmus-SMP. The authors would like to acknowledge the help and support of Csaba Józsa, András Oláh, Barna Garay and Tamás Roska at PPKE Hungary.

References

1. Alefeld, G.: On the convergence of the symmetric sor method for matrices with red-black ordering. Numerische Mathematik **39**(1), 113–117 (1982). doi:10.1007/BF01399315
2. Axelsson, O.: Iterative Solution Methods. Cambridge University Press, Cambridge (1996)
3. Bell, N., Garland, M.: Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation (2008)
4. Bolz, J., Farmer, I., Grinspun, E., Schröder, P.: Sparse matrix solvers on the GPU: Conjugate gradients and multigrid. ACM Transactions on Graphics **22**, 917–924 (2003)
5. Cantwell, C., Sherwin, S., Kirby, R., Kelly, P.: From h to p efficiently: Strategy selection for operator evaluation on hexahedral and tetrahedral elements. Computers & Fluids 43(1), 23–28 (2011). doi:10.1016/j.compfluid.2010.08.012. http://www.sciencedirect.com/science/article/pii/S00457930100
6. Cecka, C., Lew, A.J., Darve, E.: Assembly of finite element methods on graphics processors. International Journal for Numerical Methods in Engineering **85**(5), 640–669 (2011). doi:10.1002/nme.2989
7. Christen, M., Schenk, O., Messmer, P., Neufeld, E., Burkhart, H.: Accelerating stencil-based computations by increased temporal locality on modern multi- and many-core architectures. In: Proceedings of the First International Workshop on New Frontiers in High-performance and Hardware-aware, Computing (HipHaC'08), pp. 47–54 (2008).
8. Dally, B.: Power, programmability, and granularity: The challenges of exascale computing. In: Proceedings of the 25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2011, Anchorage, Alaska, USA, 16–20 May, p. 878 (2011).
9. Datta, K., Murphy, M., Volkov, V., Williams, S., Carter, J., Oliker, L., Patterson, D., Shalf, J., Yelick, K.: Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In: Proceedings of the 2008 ACM/IEEE conference on Supercomputing, SC '08, pp. 4:1–4:12. IEEE Press, Piscataway, NJ, USA (2008).
10. Fidkowski, K.J., Oliver, T.A., Lu, J., Darmofal, D.L.: p-multigrid solution of high-order discontinuous galerkin discretizations of the compressible navier-stokes equations. J. Comput. Phys. **207**(1), 92–113 (2005). doi:10.1016/j.jcp.2005.01.005

11. Filipovic, J., Peterlik, I., Fousek, J.: GPU acceleration of equations assembly in finite elements method preliminary results. Symposium on Application Accelerators in HPC, SAAHPC (2009)
12. Flaig, C., Arbenz, P.: A scalable memory efficient multigrid solver for micro-finite element analyses based on CT images. Parallel Computing **37**(12), 846–854 (2011). doi:10.1016/j.parco.2011.08.001. http://www.sciencedirect.com/science/article/pii/S01678191110
13. Göddeke, D., Strzodka, R., Turek, S.: Accelerating double precision FEM simulations with GPUs. In: Hülsemann, F., Kowarschik, M., Rüde, U. (eds.) 18th Symposium Simulationstechnique (ASIM'05), pp. 139–144.Simulation, Frontiers in (2005)
14. Hwu, WmW: GPU Computing Gems Emerald Edition, 1st edn. Morgan Kaufmann Publishers Inc., San Francisco,CA, USA (2011)
15. Johnson, C.: Numerical Solution of Partial Differential Equations by the Finite Element Method. Cambridge University Press, Cambridge (1987)
16. Komatitsch, D., Göddeke, D., Erlebacher, G., Michéa, D.: Modeling the propagation of elastic waves using spectral elements on a cluster of 192 GPUs. Computer Science Research and Development **25**(1–2), 75–82 (2010). doi:10.1007/s00450-010-0109-1
17. Komatitsch, D., Micha, D., Erlebacher, G.: Porting a high-order finite-element earthquake modeling application to NVIDIA graphics cards using CUDA. Journal of Parallel and Distributed Computing 69(5), 451–460 (2009). doi:10.1016/j.jpdc.2009.01.006. http://www.sciencedirect.com/science/article/pii/S07437315090
18. Markall, G.R., Ham, D.A., Kelly, P.H.: Towards generating optimised finite element solvers for GPUs from high-level specifications. Procedia Computer Science **1**(1), 1815–1823 (2010). doi:10.1016/j.procs.2010.04.203. http://www.sciencedirect.com/science/article/pii/S18770509100
19. NVIDIA: cuSPARSE library, last accessed Dec 20th (2012). http://developer.nvidia.com/cuSPARSE
20. NVIDIA: NVIDIA CUDA C Best Practices Guide, last accessed Aug 20th (2012). http://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf
21. NVIDIA: NVIDIA Tesla C2070 techinical specifications, last accessed Aug 20th (2012). http://www.nvidia.com/docs/IO/43395/NV_DS_Tesla_C2050_C2070_jul10_lor
22. NVIDIA: CUBLAS library, last accessed Sept 12th (2013). http://developer.nvidia.com/cublas
23. Plaszewski, P., Maciol, P., Banas, K.: Finite element numerical integration on GPUs. In: Proceedings of the 8th international conference on Parallel processing and applied mathematics: Part I, PPAM'09, pp. 411–420. Springer, Berlin, Heidelberg (2010). http://dl.acm.org/citation.cfm?id=1882792.1882842
24. Poole, E.L., Ortega, J.M.: Multicolor ICCG Methods for Vector Computers. SIAM Journal on Numerical Analysis **24**(6), 1394–1418 (1987)
25. Reguly, I., Giles, M.: Efficient sparse matrix-vector multiplication on cache-based GPUs. In: Innovative Parallel Computing (InPar), 2012. IEEE (2012). 2012, doi:10.1109/InPar.6339602.
26. Spencer, B.: A general auto-tuning framework for software performance optimisation (2011). Third Year Project Report, University of Oxford.
27. Vázquez, F., Fernández, J., Garzón, E.: Automatic tuning of the sparse matrix vector product on GPUs based on the ELLR-T approach. Parallel Computing (2011). doi:10.1016/j.parco.2011.08.003. http://www.sciencedirect.com/science/article/pii/S01678191110