



Accelerated finite element elastodynamic simulations using the GPU



Peter Huthwaite*

Department of Mechanical Engineering, Imperial College London, SW7 2AZ, UK

ARTICLE INFO

Article history:

Received 17 June 2013

Received in revised form 8 October 2013

Accepted 9 October 2013

Available online 18 October 2013

Keywords:

Finite element

Ultrasound

Elastodynamic

Graphical processing unit

GPU

ABSTRACT

An approach is developed to perform explicit time domain finite element simulations of elastodynamic problems on the graphical processing unit, using Nvidia's CUDA. Of critical importance for this problem is the arrangement of nodes in memory, allowing data to be loaded efficiently and minimising communication between the independently executed blocks of threads. The initial stage of memory arrangement is partitioning the mesh; both a well established 'greedy' partitioner and a new, more efficient 'aligned' partitioner are investigated. A method is then developed to efficiently arrange the memory within each partition. The software is applied to three models from the fields of non-destructive testing, vibrations and geophysics, demonstrating a memory bandwidth of very close to the card's maximum, reflecting the bandwidth-limited nature of the algorithm. Comparison with Abaqus, a widely used commercial CPU equivalent, validated the accuracy of the results and demonstrated a speed improvement of around two orders of magnitude. A software package, Pogo, incorporating these developments, is released open source, downloadable from <http://www.pogo-fea.com/> to benefit the community.

© 2013 Elsevier Inc. All rights reserved.

1. Introduction

Graphical processing units (GPUs) are well suited to solving highly parallel, homogeneous problems, since graphical rendering involves performing many identical calculations very quickly. GPUs utilise a single-instruction-multiple-data (SIMD) type architecture (or variants of this) [1], which allows the same instruction to be executed very efficiently many times in parallel across different data. GPUs therefore have a significant advantage over CPUs (central processing units) for solving highly parallel, homogeneous discretised engineering problems; typically GPUs can accelerate a parallelisable engineering problem by 1–2 orders of magnitude over a single core CPU implementation (see for example [2–4]).

The earliest examples of using GPUs for general purpose calculations unsurprisingly did not deviate far from traditional graphics approaches. A particularly early example from 1990 used a GPU for robot motion planning [5], and in the late 90s Hoff et al. [6] outlined the use of GPUs for generating Voronoi diagrams. The latter used OpenGL [7], an existing graphical API which became more widely used for general purpose GPU computing in the early 2000s. At this time, the potential of GPUs for general purpose programming was beginning to be realised, as noted by Trendall and Stewart [8], who provided an overview of what was considered achievable with the contemporary technology, and recognised the future potential of this field. They applied the concept to real time calculation of refractive caustics, and also included an overview of early general calculations performed on GPUs.

Further applications suitable for GPUs were developed in the following years, e.g. [9,10]. Several approaches were developed around this time to enable solution of the general linear algebra problems which commonly occur in engineering,

* Tel.: +44 (0) 207 594 7227.

E-mail address: p.huthwaite@imperial.ac.uk.

such as [11], where direct matrix solvers were developed and used to solve finite difference wave equations (amongst other problems), [12] which focused on conjugate gradient and multigrid methods to solve sparse matrix problems on the GPU, and [13], which aimed to utilise GPUs for problems where general vector processors would traditionally be used, such as matrix multiplication. As the potential of GPUs was recognised, dedicated technologies were developed to make general programming with the more advanced features of GPUs easier. An early example, from 2004, is Brook, from Stanford University, USA [14], but this was followed in later years by Nvidia's CUDA [1], ATI's Stream (formerly CTM) [15], and OpenCL [16].

In this paper CUDA is used because it provides transparent access to the GPU hardware at a low level, while minimising the programming complexity. Despite this, careful design is necessary to best exploit the features of the hardware. CUDA executes many threads in parallel, which are grouped together in blocks. Each thread can access the global GPU memory, which has high latency and is comparatively slow, and must also be accessed in a coalesced manner for maximum efficiency. Also available is a smaller region of fast, shared memory, which can be accessed by all threads within a particular block; a common approach is to load a section of data into the shared memory, perform calculations using this data, then save the resultant data back to the global memory.

The architecture is well suited to performing explicit time domain simulations, where the variables at one time step can be explicitly calculated from their known values at previous time steps. Typically, there is substantial sparsity to the problem, so that the value at one position only depends on previous values at nearby locations, rather than the entire domain. The fast shared memory for each block, available using CUDA, is ideal to exploit this; data from within a small region of the domain can be loaded in to the shared memory, then many values at the next time step can be calculated without requiring any data from outside the block.

A problem arises when a calculation needs to be performed which requires access to data from a neighbouring block. A degree of overlap between the blocks must be implemented to allow such data to be accessed, and this must be performed in an efficient way to avoid unnecessary global memory loads. GPUs have been applied extensively to problems where the domain is discretised via a structured grid (e.g. [17–19]), which results in uniform relationships between the variables. This greatly simplifies memory addressing and aids the design of an efficient solution to allocate data to each block.

Our focus is instead on problems that are discretised using a more general “free”, unstructured mesh, which allows it to 1) be refined locally if necessary and 2) conform to complex boundaries without causing the “staircasing” which occurs with uniform meshes [20–22]. The challenge is how to subdivide the unstructured meshes into separate partitions suitable for each block and arrange the data in memory in an efficient manner to enable the calculation to be performed quickly.

The use of unstructured meshes is widespread with the finite element (FE) method. The potential of GPUs for such FE problems was proposed by Fan et al. [23], and an OpenGL implementation of electromagnetic time domain FE calculations was described by Liu et al. [24]. Another early study of FE solutions, this time focusing on addressing the inaccuracies caused by the contemporary single-precision-only GPUs was undertaken by Göddeke et al. [25]. This group has done extensive further work in the area of FE on GPUs [26–30], including incorporating GPU technologies into their package FEAST (Finite Element Analysis and Solutions Tools) [31]. FEAST has an interesting approach to the challenges associated with solving problems on unstructured FE meshes using parallel architectures, subdividing the domain into regions of locally structured sections, which enables better performance. There have also been several applications of GPUs to non-linear elastic FE problems, [32] for soft tissue modelling, and [33] for large elastic deformations.

The challenge of suitably arranging data in memory for explicit time-stepping finite element problems has, however, received limited attention in the literature. Komatitsch et al. [34] ported an unstructured mesh model of the earth to CUDA; their approach was to use high-order spectral elements of five nodes in each of three dimensions. The resulting 125-node elements fitted well within a block of 128 threads, which could then be efficiently arranged in memory; while this is an elegant solution it is unlikely to be practical for the general case, particularly when considering lower order problems. Klöckner et al. [35] recognised the problem for the more general case, where many elements must be allocated to each block; a partitioning scheme was therefore required to divide the mesh into blocks in an efficient manner. It was recognised that general purpose partitioners such as those developed for parallel computing [36] were rarely suitable, since the specific nature of the hardware configuration requires particular limits on size which are challenging to implement. A simple ‘greedy’ partitioner was presented as an alternative; however, while such an approach is quick and reliable, for the majority of meshes the subdivision will not be particularly close to optimal, which will slow down the solution stage. Additionally, little attention has been focused on the development of memory arrangement systems once the domain has been arranged into blocks.

This paper presents a more refined solution to the mesh partitioning problem for unstructured finite element meshes of 2D first order (i.e. 3-noded) triangular elements. This method is used for purely explicit time-stepping methods, i.e. solutions with diagonal lumped mass and damping matrices. Also developed is a technique to arrange the data within each block to allow efficient transfer between adjacent blocks. A new open source package, Pogo, has been developed to make the methods in this paper readily available; Pogo is a GPU implementation for the solution of elastodynamic problems discretised with linear finite elements in space, and solved with explicit time steps. This problem is applicable in a number of fields such as non-destructive testing [37–40] and seismology [41], and the source code and binaries of Pogo are made available at <http://www.pogo-fea.com/> to benefit the community. Other wave problems can be solved in a similar way, making the methods discussed here applicable to acoustic and electromagnetic waves with little modification. Furthermore, the concepts are applicable to solve general discretised engineering problems using the GPU.

Section 2 provides an overview of the relevant aspects of the finite element method and how it is implemented on the GPU. Section 3 discusses mesh subdivision and memory arrangement approaches to allocate the nodal data into memory, allowing the simulation to be performed in an efficient manner. Section 4 presents some examples of the software applied to realistic problems from non-destructive testing, geophysics and engineering.

2. Implementing the finite element algorithm on the GPU

2.1. Finite element theory

The finite element method for discretising elastic problems is well known so will not be repeated here; readers are referred to [42] which gives a good overview of the technique. The following is a derivation of the well established explicit time domain finite element method for elastodynamics, and is included because the details are important for the implementation on the graphics card. We note that similar equations can be formed using the finite element method for acoustic, electromagnetic and other forms of wave problems. Upon discretising the problem, the equations become

$$M\ddot{U} + C\dot{U} + KU = F \quad (1)$$

where U , \dot{U} and \ddot{U} are vectors of the displacement, velocity and acceleration respectively, with each term in the vector corresponding to a particular degree of freedom of the model. M , C and K are the mass, damping and stiffness matrices, with $NDOF \times NDOF$ terms; $NDOF$ is the number of degrees of freedom in the model. F is a vector describing the force applied at each degree of freedom.

A standard finite difference scheme is used to step in the time domain. Using this, the problem can be written in terms of the displacement vectors at the previous, current and next time steps, described by U_{prev} , U_{curr} and U_{next} respectively

$$M \frac{U_{next} - 2U_{curr} + U_{prev}}{\delta t^2} + C \frac{U_{next} - U_{prev}}{2\delta t} + KU_{curr} = F \quad (2)$$

which is known to have an error of the order $(\Delta t)^2$, where Δt is the time step [43]. This can be rearranged to

$$U_{next} = \left(M \frac{1}{\delta t^2} + C \frac{1}{2\delta t} \right)^{-1} \left[F + \left(C \frac{1}{2\delta t} - M \frac{1}{\delta t^2} \right) U_{prev} + \left(M \frac{2}{\delta t^2} - K \right) U_{curr} \right], \quad (3)$$

which is an explicit scheme since all terms on the right hand side are known. Eq. (3) is optimised as

$$U_{next} = F' + C'U_{prev} + K'U_{curr} \quad (4)$$

where we have introduced the modified matrices

$$K' = \left(M \frac{1}{\delta t^2} + C \frac{1}{2\delta t} \right)^{-1} \left(M \frac{2}{\delta t^2} - K \right),$$

$$C' = \left(M \frac{1}{\delta t^2} + C \frac{1}{2\delta t} \right)^{-1} \left(C \frac{1}{2\delta t} - M \frac{1}{\delta t^2} \right),$$

and the vector

$$F' = \left(M \frac{1}{\delta t^2} + C \frac{1}{2\delta t} \right)^{-1} F.$$

Since K' , C' and F' remain constant for all calculations, they can be calculated initially and reused for all time steps.

The widely used simplification of lumped mass and damping matrices is used, so that M and C are diagonal. Here, simple direct lumping is used, so the mass at each node is given by multiplying the element area by density, then dividing by the number of nodes. This greatly simplifies several of the calculations. Firstly the inverse term

$$\left(M \frac{1}{\delta t^2} + C \frac{1}{2\delta t} \right)^{-1}$$

becomes straightforward to calculate, and will itself be diagonal. Because of this, C' will also be diagonal, and the sparseness of the stiffness matrix K and the force vector F will be maintained in K' and F' respectively.

The FE computational kernel calculates U_{next} from Eq. (4) for a specified node. To do this, it needs to load in all necessary data from global memory to local memory, perform the calculation, then store it back to global memory. The memory access is most straightforward if there is a one-to-one mapping between the memory accessed and the threads, so that no data needs to be shared between different threads. We consider calculating the i th nodal value, U_{next}^i . The first term, F' , only requires access to the i th term, F'^i , which is straightforward. In the second term, $(C'U_{prev})^i$ must be calculated; due to C' being diagonal, this simply becomes $C'^{ii}U_{prev}^i$. C'^{ii} and U_{prev}^i are, as before, straightforward to obtain since they only require

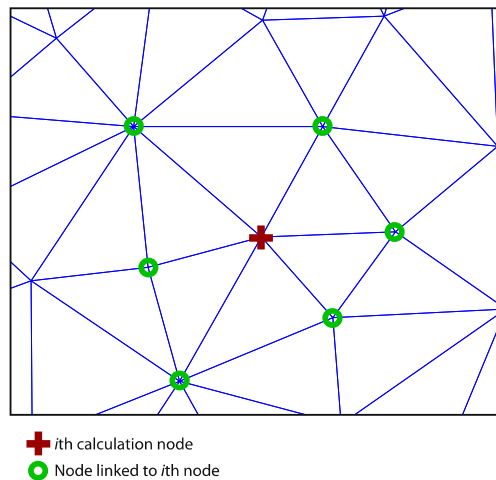


Fig. 1. Finite element node linking.

access to values at node i . The final term, $K'U_{curr}$, however, is less straightforward to calculate, being dependent on other nodes, and hence the sparsity of the effective stiffness matrix K' .

The stiffness matrix defines the force applied at a particular degree of freedom necessary to cause a unit displacement at another degree of freedom. In the finite element formulation, the stiffness matrix is formed by summing stiffness matrices from each separate element; therefore the global stiffness matrix is only non-zero where two nodes are directly linked by an element. Fig. 1 illustrates a particular node associated with the i th degree of freedom, and also marks the linked nodes – which will have non-zero stiffness values associated with them in the i th row of the effective stiffness matrix K' . Therefore, to calculate the i th value of $K'U_{curr}$, the displacement values U_{curr} of all linked nodes must be retrieved, and they must be multiplied by the corresponding coefficients stored on the i th row of K' . For efficient calculation, the nodes (and their associated degrees of freedom) must be arranged appropriately in memory; the next section discusses the architecture of the graphics card and its implications for this problem.

2.2. GPU architecture

A thorough analysis of the Nvidia GPU architecture from a programmer's perspective, along with an explanation of the CUDA technology with which to access it, is given in the Nvidia CUDA C Programming Guide [1]; rather than repeating the existing work, the focus of this section is to provide sufficient knowledge to allow understanding of the finite element implementation developed in this paper.

CUDA's approach to performing calculations on the GPU is through the use of kernels. A CUDA kernel is effectively a function, written in a subset of C++03, which is executed in parallel by many threads. The threads used to execute a kernel are arranged in blocks, which are typically up to 512 threads in size. The threads within each block are usually arranged in a 2D matrix. The blocks are then themselves arranged in a 2D grid. This is illustrated in Fig. 2. The programmer can specify both the dimensions of the blocks and the grid of blocks when requesting that a particular kernel is executed on the GPU. The kernel is provided with an index indicating which block it is in and which thread within the block it is; the kernel can then perform its calculation on the relevant section of data.

The threads within a block are executed in groups called warps, each consisting of 32 threads. Each warp will execute a single common instruction at one time, so the best performance can be achieved when all the threads within a warp need to execute the same instruction. In some cases, for example, when encountering an 'if' statement which only applies to some of the threads within the warp, the warp can diverge, which means that some of the threads must remain idle while the instruction is executed for the appropriate part of the warp.

The graphics card consists of several levels of memory; here, just the two most important are considered: (1) the global (off-chip) memory and (2) the shared (on-chip) memory. The global memory is a large region of memory – typically in the region of 1–6 GB. This can be used to store all the problem data, but has relatively low bandwidth. Additionally, there are rules specifying how the global memory should be accessed in order to achieve optimum transfer speeds; the actual details of this are dependent on the 'CUDA compute' capabilities of the card [1]. For the majority of modern GPUs available for purchase the CUDA compute version is typically above 2.0. For these cards, the global memory can only be accessed by 32, 64 or 128 byte loads; the number of global memory loads a warp of threads must make is equal to the total number of such loads needed to access all the data. This presents a strong motivation to arrange the data so that warps only require data from local regions of memory, ideally loading entire 128 byte lengths simultaneously. Additionally these cards have caches which can reduce the amount of global memory access, again provided there is a degree of locality in how the memory is accessed.

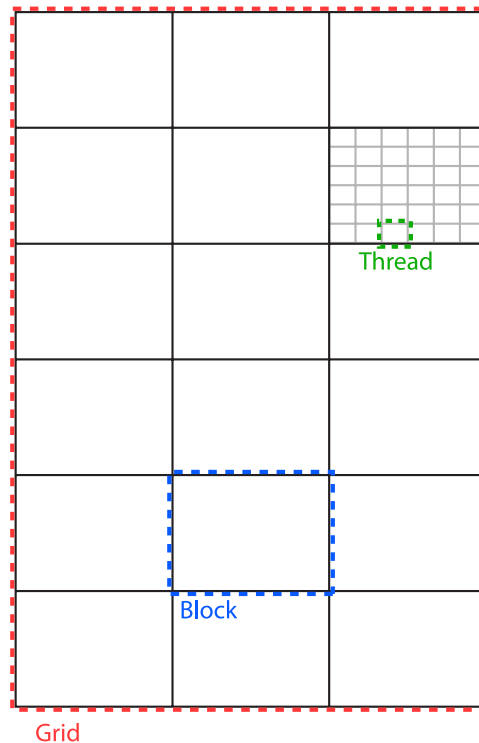


Fig. 2. Arrangement of threads under CUDA, showing the grid, block and thread hierarchy.

Shared memory stores data on the chip while the threads are in existence, is much faster (i.e. higher bandwidth and lower latency) than global memory, and does not need to be accessed in 32 byte loads. The typical principle of a CUDA kernel is to copy data as efficiently as possible from the global memory to local memory, perform a series of calculations in local memory, then copy the data back to global memory. In the case of an explicit time domain algorithm such as is considered in this paper, this process would be repeated for each time increment.

2.3. Time domain FE on the GPU

A scheme has been developed to perform the time domain finite element calculations on the GPU. A single kernel is used, and a thread is assigned to each node in the domain. The nodes in the domain must be subdivided into separate blocks; performing this subdivision is discussed in the next section.

There is a compromise in the choice of block size; a large block will have a smaller fraction of boundary nodes, so relatively few reads from adjacent blocks will be necessary. The smaller resource requirements of smaller blocks, however, allow the GPU multiprocessors to be assigned more than one block at once and hence perform calculations with one block, for example, while data for another block is loading. A relatively large block size of 32×16 threads is chosen to take advantage of the first consideration, although future work could be performed to establish whether additional performance could be achieved with alternative block sizes. Nodal data is assigned to global memory such that the x dimension of the block is fastest, the block number is the middle, and the y dimension of the block is slowest. This keeps the memory aligned so that loading in the data from each block can be performed in a coalesced manner.

Fig. 3(a) demonstrates how the FE mesh can be subdivided into blocks, and Fig. 3(b) shows the allocation of nodes in the blocks to the regions of global memory. Fig. 3(c) shows the local memory for one block. The first 16 rows are simply loaded directly from the appropriate area of global memory, shown in Fig. 3(b). This enables the data from the $32 \times 16 = 512$ nodes to be loaded. The remaining rows are loaded from other blocks. Loads from adjacent blocks of 32, 16 and 8 nodes are defined in recognition that the common boundaries of adjacent blocks typically have a wide variety of lengths.

The numbers of each of these loads are chosen initially based on a structured mesh, formed from a uniform grid of squares, each split diagonally to form two triangular elements. For convenience, we can subdivide this into aligned partitions of size 32×16 nodes. Each partition will need to access 32 nodes from above, 32 nodes from below, and 16 nodes from each of the left and right. In the corners, a node from two of the four diagonals is required (all four would be necessary if 4-noded square elements are used – the diagonal division for triangular elements removes the diagonal linking for two of the corner nodes). Thus 2×32 -node, 2×16 -node, and 2×8 -node loads need to be defined in this case, which provides a starting point for a general solution. Clearly with an unstructured mesh, the loads will need to be more flexible, so more smaller loads need to be added. The numbers are refined by testing with some typical free meshes. On one hand the aim

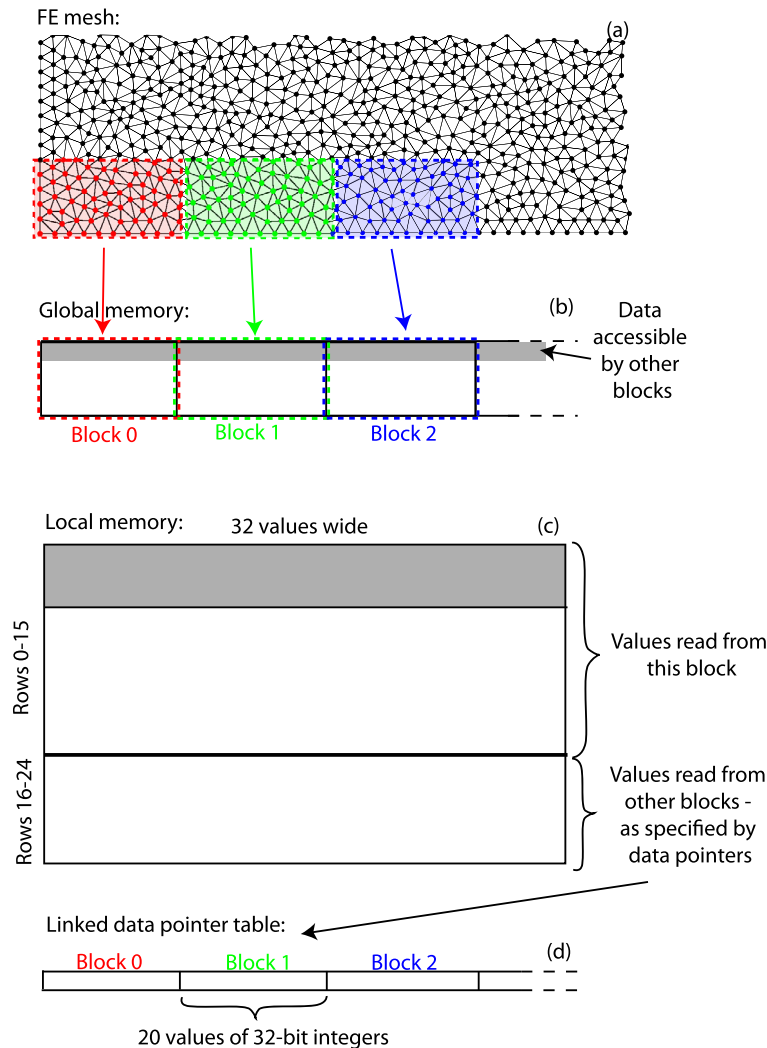


Fig. 3. (a) Schematic of mesh subdivision into separate blocks and (b) assignment of data for each block in global memory. (c) Local memory layout, showing region loaded directly from the block of interest, and the memory at the end filled with data from other blocks. The linked data pointer table (d) indicates which data from which other blocks is to be loaded into the 20 banks at the end of (c).

is to have as few loads as possible since loading extra data slows the algorithm down, but more loads adds flexibility and makes it more straightforward for the partitioner to establish a suitable memory arrangement scheme. Ultimately a set of 20 loads are defined for the 2D problem with 3-noded triangular elements. These are allocated as 2 of 32 nodes, 10 of 16 nodes and 8 of 8 nodes.

The linked data pointer table of Fig. 3(d) is read by each block and contains 20 32-bit integers per block. These numbers indicate where in memory each of the 20 memory banks should load from. Of each of these numbers, the most significant 24 bits indicates the block from which to load the data, and the remaining 8 bits indicate a starting location within the block. The first 4 bits indicate the row, with the most significant 2 bits of the value being unused at present, and the remaining two bits specifying which of the first 4 rows should be loaded. The second set of 4 bits similarly has its higher two bits unused, then the second two bits are multiplied by 8 to give a starting column. The unused bits can potentially be used for alternative future memory layouts which require additional location information within the block.

Once all the data from all the nodes and adjacent nodes is loaded into shared memory, the calculation can be performed. Each node in the block has enough information about the previous values within the shared local memory of Fig. 3(c) to enable the next time step to be calculated. It is also necessary to load in stiffness values; it is assumed that there are up to L nodes linked to each node. The parameter L varies depending on the element type and number of dimensions in the model. For the 2D linear triangular elements $L = 12$ forms a good upper limit. It is more efficient for the software to expect the same number of links for each node, then have some which are unused, than having different numbers of links for each. Under the latter scenario, each node would have to store a reference to the initial memory access location, along with the number of links to be loaded. Then there are the challenges of aligning the data in memory for coalesced loads. These

issues are likely to significantly affect the speed of the algorithm. It should be noted that for the majority of well-formed meshes the number of linked nodes is unlikely to vary significantly between nodes. A fixed limit is chosen for these reasons. Since there are 2 degrees of freedom per node, there are therefore $L \times 2 \times 2$ values from the stiffness matrix to be loaded in per node, and also L 'pointer' values specifying to which of the 32×25 nodes accessible within the block the stiffness values correspond. The sparse stiffness values are therefore stored in a set of four vectors with $L \times 512 \times n_{\text{blocks}}$ terms, and the corresponding pointer vector of the same size. Any unused terms have the pointer vector set to -1 .

The scheme presented here provides a general method to access data efficiently for unstructured meshes in the GPU block structure. However, we have not yet given consideration to how a general mesh can be arranged in memory to exploit this configuration; a suitable method is developed in the next section.

3. Mesh subdivision algorithm

There are two stages necessary to fit a mesh to the memory arrangement discussed in the previous section. Firstly, the mesh must be subdivided into (up to) 512-node sections which can each be assigned to a block. Secondly, the memory within each block must be arranged according to the scheme discussed in the previous section.

This problem is extremely challenging because there are only finite resources available to each block. For example, since only the first 4 rows, or 128 nodes, of a block can be accessed by other blocks, the boundary length of each block must be less than 128. Since we can only complete 20 loads from other blocks, each block must be bounded by fewer than 20 blocks, and we are restricted in how many nodes we can load from each of them.

The approach is therefore to partition the mesh as neatly as possible, attempting to minimise the communication between adjacent blocks. The memory is then arranged for each block; if it is found that this is not possible, then the partition arrangement is locally adjusted and another attempt is made. This is repeated until a solution is found.

The ultimate aim of the algorithm is to minimise the number of blocks, since this directly affects the runtime of the solver. This means that each block should contain as many nodes as possible – ideally 512. An efficiency value is defined as $\eta = n_{\text{nodes}} / (n_{\text{blocks}} \times 512)$, with n_{nodes} being the number of nodes in the domain and n_{blocks} being the number of blocks; the target is to obtain a value close to 1.

3.1. Partitioning

Domain partitioning has been widely studied in order to balance loads across multiple CPU processors. [36] summarised the performance of several of these algorithms. However, the needs of the Pogo partitioner are quite specific, and such general algorithms are unlikely to produce an efficient result. Two partitioning schemes are considered in this paper. The first is the simple 'greedy partitioner'; the algorithm follows that of [35]. The second is a more efficient partitioning scheme – the 'aligned partitioner' – which has been developed to subdivide the mesh into neat aligned blocks.

The aligned partitioning algorithm is given as follows; these steps are illustrated in Fig. 4.

1. Find nodes on outer boundary of domain
2. Subdivide nodes on outer boundary into lengths approximately $B = 32$ nodes long – each length becomes the start of a new block
3. Advance each block by a row
 - (a) find all nodes linked to a block which are unassigned to blocks
 - (b) assign them to the current block
4. Repeat the advance until $R = 20$ rows added
5. Loop through blocks to get sizes approximately correct:
 - (a) If block size is less than $S_l = 390$, join to an adjacent block
 - (b) If block size is more than $S_u = 1000$, split into two blocks
 - (i) Find the two ends of the block
 - (ii) Advance blocks from both ends until they meet in the middle
 - (c) Repeat until all blocks satisfy the requirement $S_l < \text{block size} < S_u$
6. Remove all blocks with the exception of the first row of each block
7. Find the smallest block and advance that by one row; stop if the block reaches 512 nodes
8. Repeat 7 until all blocks are filled with 512 nodes
9. Start a new layer of blocks by advancing each block by a row and assigning this new row to a new block
10. Repeat from 3 until all nodes are assigned to blocks

The 'advance' process of the algorithm is similar to the advancing wavefront-type methods introduced by Cuthill and McKee [44]; these approaches aim to order the nodes so as to minimise the bandwidth of the sparse stiffness matrix. By performing a similar 'advance' we aim to arrange the nodes into blocks to minimise interdependence, which is similar to minimising the bandwidth. The approach is also related to advancing front meshing algorithms [45–47], where a mesh is generated by starting at the domain boundary and adding successive layers of elements to fill the domain.

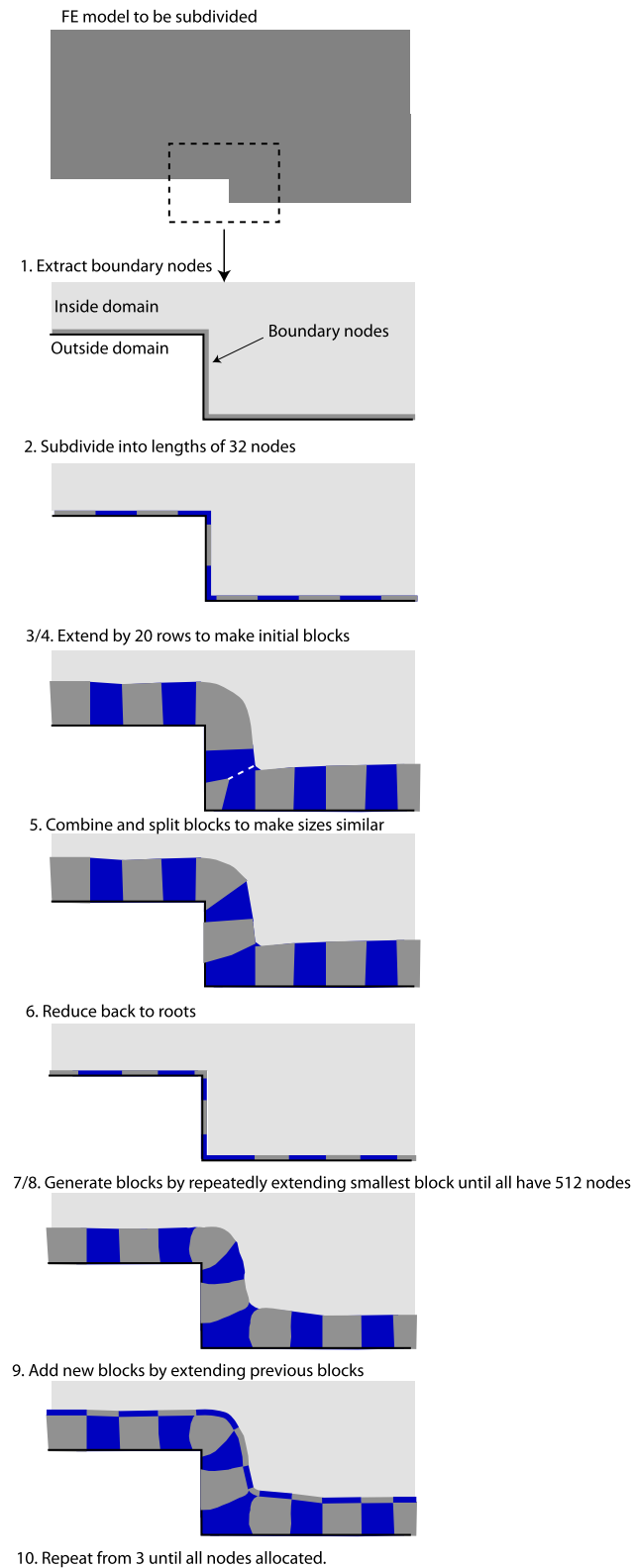


Fig. 4. Stages of the mesh subdivision algorithm.

An interesting feature of this algorithm is the ‘double advance’. If we have a concave external boundary, as each row is added, the row will get longer and hence the block will become bigger compared to the others. If the boundary is convex, the block will become smaller. The initial advance tests whether blocks are on particularly concave or convex boundaries by checking their sizes. If they are too small, they will be combined with adjacent blocks, and if they are too big they will be split. When the blocks are subsequently reduced back to the first row, these will form a much neater base for the actual advance.

The constants here are chosen to fit with the 512 nodes/block scheme, and can be adjusted if necessary for other configurations. The initial subdivision of the boundary into lengths of $B = 32$ is based on a block shape of 32×16 ; for an arbitrary block size to obtain an aspect ratio of around 1:1 to 2:1, a suitable range for B should be $\sqrt{n_{npb}} < B < \sqrt{2n_{npb}}$ where n_{npb} is the number of nodes per block. A choice of $R = 16$ might be sensible for the ‘test’ advance, since that would give on average 512 nodes per block. However, the block is extended slightly further than this, by making $R = 20$ to give an increased weighting to split or join blocks which will require this action in later layers anyway. In a general implementation, adding an extra 25% as done here would give $R = n_{npb}/B \times 1.25$. Finally, the limits S_l and S_u need to be established. A mean value can be calculated as $S_m = R \times B$. Lower and upper limits can be based around this; if the shape of the blocks is relatively unimportant, the acceptable range can be widened, or it can be tightened to ensure the blocks fall closer to the required size. It has been found that the values used here fit well with the algorithm, so it is suggested for a general case to use $S_l = 0.6 \times S_m$ and $S_u = 1.55 \times S_m$. Clearly for best performance it is suggested to fine-tune these constants by testing them out in the algorithm.

Sometimes domains include internal boundaries; this is particularly important in non-destructive testing applications, for example, where internal defects such as cracks or voids can exist. The partitioning algorithm has a couple of modifications to account for the situation where the ‘advancing front’ of partitions encounters such a boundary. The initial advance will be self-limiting, i.e. the partitions will simply only advance as far as they can until they interact with the boundary. As with the convex external boundary, the restriction in size, if significant, will automatically encourage this partition to be combined with a neighbour, to enable a better starting point; no modification needs to be made here.

For the second, ‘final advance’, there are a couple of scenarios which need to be accounted for. Firstly, any partition which becomes completely trapped by a boundary such there are no adjacent ‘unpartitioned’ nodes is deactivated; from this point it will not try to advance any further or produce any ‘child’ partitions in the next layer. The second possibility is that the advancing front encounters an acute internal boundary which splits a single partition. To account for this, when child partitions are produced, a check is performed to verify whether all nodes within a single child are contiguous; if they are not then they are separated and allocated to different partitions. These modifications enable the algorithm to correctly account for internal defects.

3.2. Memory arrangement

Having partitioned the nodes into separate subdomains, the data must be arranged in memory. Firstly, the boundary of each block must be arranged in the first four rows of the global memory. This must be arranged so that each adjacent block can access data with as few loads as possible.

The strategy adopted is to order the boundary nodes by looping around the boundary from one point. This ensures that, with the exception of the start-end point, all nodes linked to a particular node will be contiguous in memory. The start-end point is moved around the boundary until it reaches a join between two blocks. Clearly, since the linked nodes are split over up to 4 lines, there will be additional memory breaks; possibly this could be improved by ‘wrapping’ to match the memory breaks to the joins between different blocks. However, such a solution is likely to be complex so is not pursued at present.

Once the boundary nodes of each block have been arranged in the first rows of the global memory, the linked data pointer table discussed in Section 2.3 needs to be established to provide links between blocks. For each block, the initial stage is to determine which blocks are linked to it, and from which locations within this block nodes need to be loaded. This principle is illustrated in Fig. 5(a) and (b). Due to the memory loading scheme introduced in the previous section, each 8-node long section must be loaded in its entirety; Fig. 5(c) simplifies the problem to demonstrate which of the sections must be loaded.

Having established which sections need to be loaded, the different sized loads available (2×32 -nodes, 10×16 -nodes and 8×8 -nodes) need to be allocated. The approach is to use the larger loads as efficiently as possible for the contiguous sections, then use the smaller loads for the remainder. The 32-node loads are allocated according to the priorities in Fig. 5(d). Clearly the best use is to load an entire line; this therefore has the highest priority, whereas the lowest priority is given to loading a single 8-node section with a 32-node load.

The algorithm is:

1. Take one of the 32-node loads
2. Look at the first row of the first surrounding block
3. If it has priority 1 (see Fig. 5(d)), assign the load to this row, and return to step 1 to take the next 32-node load
4. Otherwise proceed through all the rows of all the surrounding blocks until a priority 1 row is found

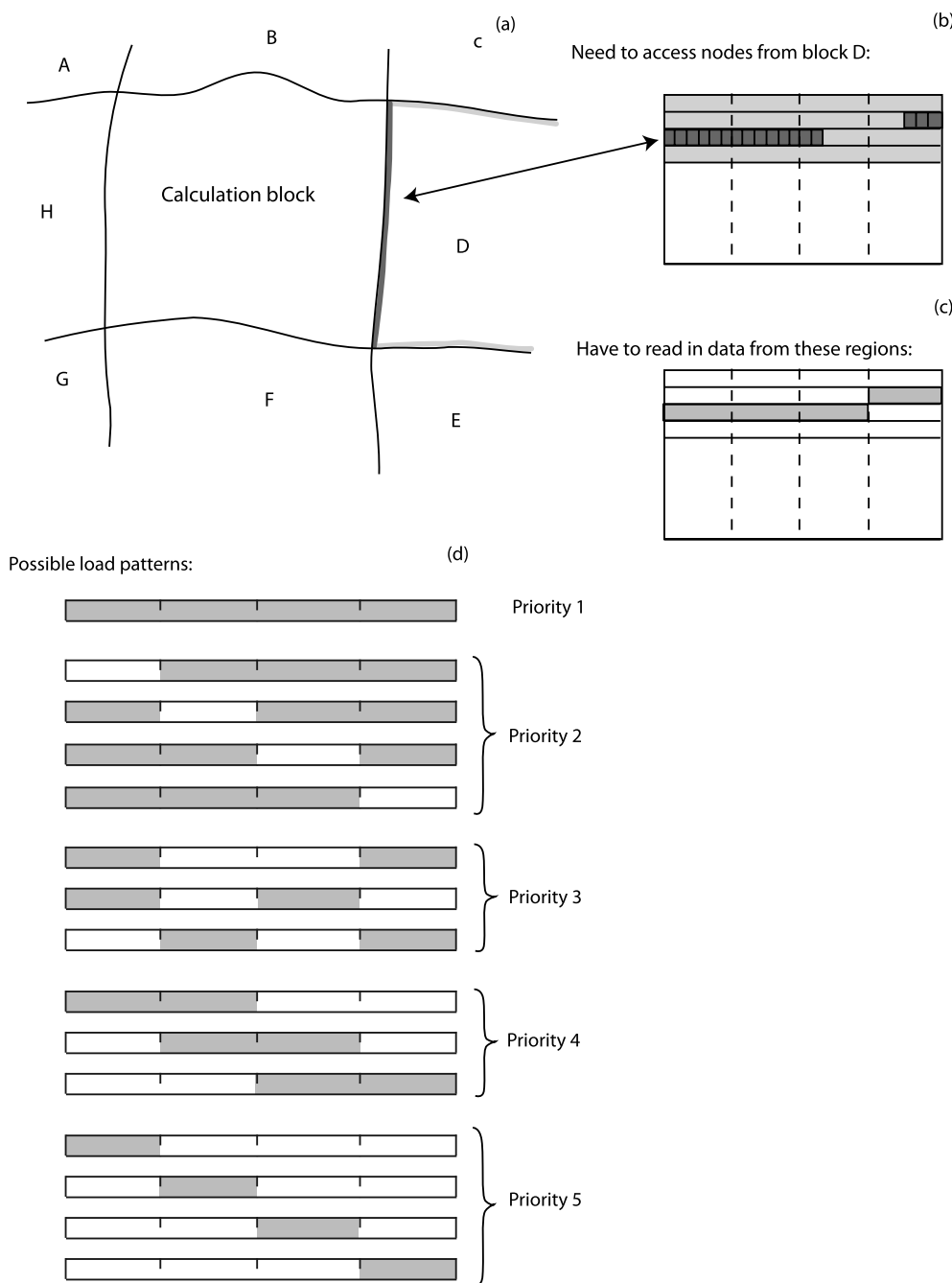


Fig. 5. Arranging the links between the blocks. (a) gives a typical block layout, showing a block and the 8 surrounding blocks labelled A–H. (b) shows which nodes need to be accessed from linked block D. (c) then shows how this is simplified into accessing particular 8-node long sections of data. (d) presents all the possible load patterns within a single row; these are put into priority of preference to be allocated to a 32-node load.

- 5. If none are found, do the same with priority 2
- 6. Repeat for all priority levels until both 32-node loads are taken

Once both 32-node loads are allocated, the algorithm moves to the 16-node loads. The same algorithm is used, except here the priority is simply to load in two consecutive 8-node sections. If this is not possible, the algorithm loads in isolated single 8-node sections, effectively wasting half of the load. Finally, the 8-node loads are allocated to any remaining 8-node sections.

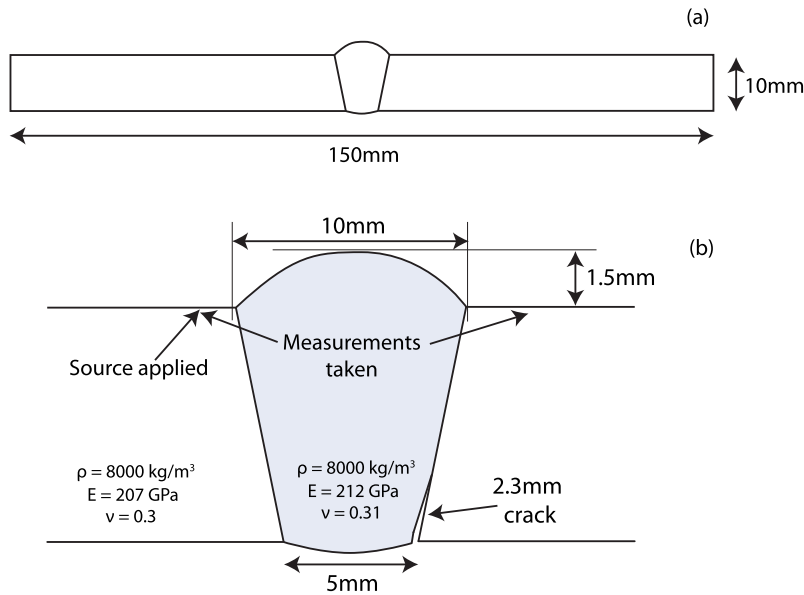


Fig. 6. Diagram of weld finite element model. (a) presents the plate containing the weld. (b) gives more detail of the weld itself, including dimensions.

If the algorithm has allocated all the available loads yet there are some sections of memory that still need to be allocated, the block is considered to have ‘failed’ the memory allocation algorithm and the partitioning needs to be adjusted. The approach is to split any block which fails into two (which reduces efficiency by adding extra blocks), and redo the memory mapping algorithms for all surrounding blocks. This process is repeated until all blocks in the domain are able to access the nodes they need.

Clearly, blocks are more likely to pass the memory allocation algorithm if they are neatly arranged and aligned with each other. This is why the partitioning scheme of the previous section is important: it is vital to generate a neat initial partition so that the majority of blocks can have data allocated to them without resorting to splitting. It is expected that a higher proportion of blocks produced with the greedy blocker will be poorly formed compared to the aligned partitioner, and therefore efficiency associated with the greedy partitioner is likely to be lower.

The next section tests the partitioning algorithms and solution performance for three example problems.

4. Examples

In this section three example models are presented. These models were run on the CPU using a typical commercial software package, Abaqus, and on the GPU using Pogo, the finite element package developed in this paper. Comparisons are performed to check that Pogo produces the same result as Abaqus, and the speeds of the two software packages are studied and compared to the theoretical maximum performance of the CPU and GPU.

In all models, the default solution settings were used when running the Abaqus job, i.e. the Abaqus packager was run in single precision but the explicit solver was run in double precision. Pogo was run primarily in double precision, with single precision results included for comparison.

The explicit component of Abaqus version 6.12 was used. The equations solved [48] match those used in Pogo, outlined in Section 2.1, including in the mass and damping lumping approaches. This was run on a single core of an HP Z600 workstation, with dual quad core processors. The processors were Intel Xeon E5530 processors running at 2.4 GHz. The Pogo solver was run using an Nvidia GeForce GTX580 GPU, a high end gaming graphics card. It is recognised that this GPU is around 1–2 years newer than the CPU, so the relative performance of the GPU would be expected to be better. An analysis of the relative hardware capabilities is included when discussing the results.

4.1. Weld inspection example

Non-destructive testing relies extensively on ultrasound for subsurface inspection. A common application is the inspection of welds to check whether there are any cracks present at the boundary between the weld and the base material, which could lead to a failure. Ultrasound is passed into the weld and the measured response is processed to determine whether or not cracks are present.

A typical weld model was set up using the Abaqus CAE graphical user interface. A schematic diagram of the model is presented in Fig. 6. This was meshed with 3-noded linear triangular elements using the Abaqus unstructured triangular

Table 1
Finite element parameters for the different models.

Property	Weld model	Gear model	Geophysical model
Signal type	Hann	Hann	Gaussian
Frequency	5 MHz	500 kHz	N/A
Signal length	5 cycles	2 cycles	N/A
Element size (approx. side length)	40 μm	80 μm	5 m
Simulation time	40 μs	1 ms	5 s
Simulation time increment	1 ns	2 ns	0.2 ms
Number of time increments	40 000	500 000	25 000
Degrees of freedom ($\times 10^6$)	1.92	2.89	2.02

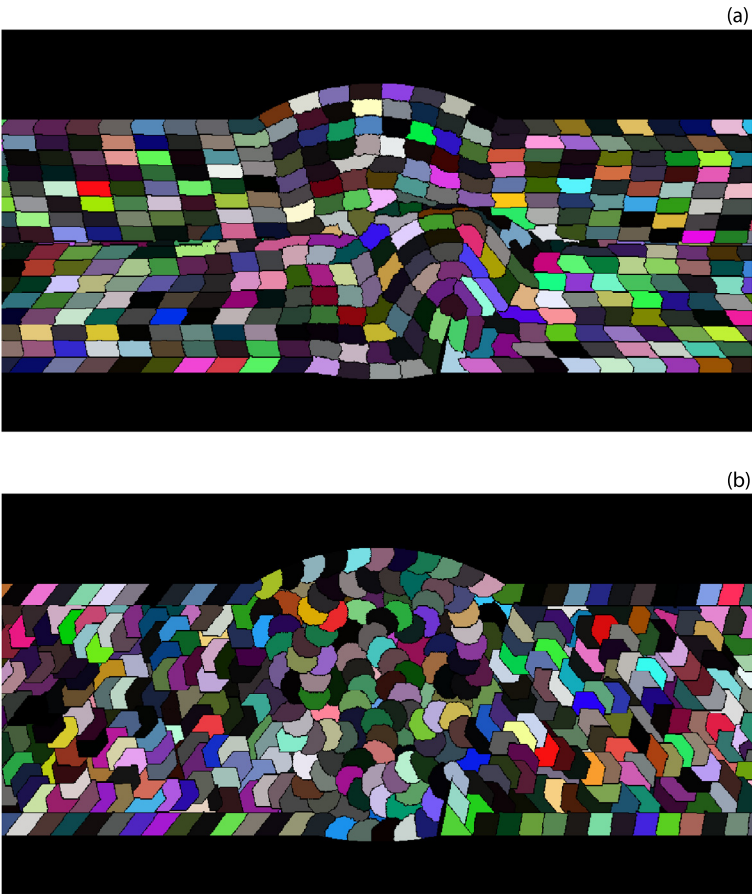


Fig. 7. Different blocking arrangements for a section of the weld finite element model outlined in Fig. 6. (a) is the aligned blocker and (b) is the greedy blocker.

mesher. The properties of the model are given in Table 1; the 2D problem was modelled as plane strain. The source was excited as a force at 45 degrees downwards to the right.

An input file translator has been written to generate a Pogo input file from an Abaqus input file. Clearly, since Pogo only supports a subset of Abaqus’ functionality, the translator can only process certain specific keywords appropriate to explicit time domain simulations. The two partitioning approaches – the aligned and the greedy partitioners – were both applied to the Pogo input file to generate a Pogo block file, a file indicating which nodes were allocated to which block and their locations within the block. Each block file was passed to the Pogo solver, along with the input file, to calculate a solution.

The two blocking arrangements are presented in Fig. 7. The aligned blocker of Fig. 7(a) achieved an efficiency of 95% according to the definition in Section 3, while the greedy blocker in Fig. 7(b) achieves 88.3%. The aligned blocker result is clearly a neater solution; the main inefficiency appears at the line where the two advancing fronts of blocks meet at the horizontal centre of the plate. The greedy blocker result is less organised, with several significantly distorted blocks; such a solution typically requires the second stage, where the memory is arranged within the blocks, to split more blocks and hence reduce efficiency further.

Table 2

Comparison of run-times for the different models between Pogo running on the GPU, and Abaqus, a typical CPU alternative. All times are in seconds. The pre-processor time for Abaqus includes the 'pre' and 'packager' stages; for Pogo the pre-processor time is any time taken by the solver which does not include the explicit time-stepping stage. For Pogo, the two cases given are for the aligned and greedy partitioners, and the times in brackets are single precision times; all the other times are double precision.

Model	Solution type	Blocker efficiency (%)	Block time	Pre-processing time	Explicit stepping time	Total time
Weld	Abaqus	N/A	N/A	125	16 282	16 407
	Pogo (aligned)	95.0	23.4	23.6	75.3 (43.7)	122.0
	Pogo (greedy)	88.3	1.9	22.6	78.4 (43.1)	102.9
Gear	Abaqus	N/A	N/A	217	367 253	367 470
	Pogo (aligned)	96.3	50.0	32.4	1476 (822)	1558
	Pogo (greedy)	81.7	6.9	37.5	1588 (925)	1633
Geo.	Abaqus	N/A	N/A	147	12 477	12 624
	Pogo (aligned)	99.7	16.7	10.1	49.4 (27.3)	76.2
	Pogo (greedy)	84.4	2.5	10.4	54.6 (29.8)	67.5

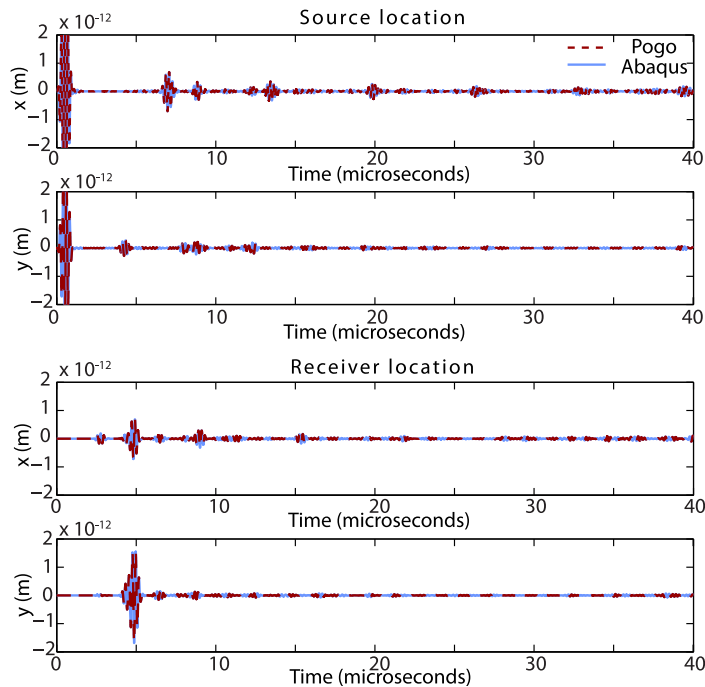


Fig. 8. Comparison between displacement time traces for Pogo and Abaqus at the points marked in Fig. 6.

Table 2 presents the run times for the various models, showing how the explicit time-stepping stage for the weld model can be solved by Pogo in 75.3 s with the aligned blocker. Fig. 8 compares the Abaqus and Pogo time traces at the measurement locations marked in Fig. 6. It is clear that the traces match extremely well, to the extent that they are visibly indistinguishable. Fig. 9 calculates the error in displacement between Pogo and Abaqus for the two measurement locations, illustrating that this is around five orders of magnitude below the actual values.

4.2. Mechanical example

Explicit time domain solutions can be used for studying vibration behaviour of mechanical components. In this example, a gear-type component is excited at one of its teeth and the resulting time trace monitored at a point on the far side of the model.

Fig. 10 presents the model. The material properties are the same as for the steel base material for the weld example of Section 4.1. The model parameters are detailed in Table 1; the excitation force is applied to all nodes on the excitation surface in a direction vector $(-1, -0.8)$ in the horizontal and vertical directions respectively. This model is of particular interest since it contains several internal boundaries and will hence test the performance of the partitioning algorithms when dealing with these.

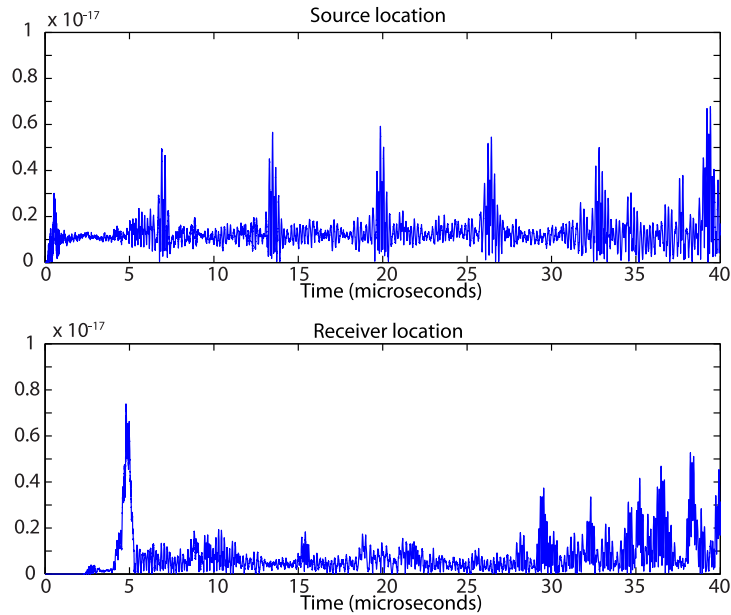


Fig. 9. Discrepancy between time traces for Pogo and Abaqus. Discrepancy is calculated as $\sqrt{(u_x^{pogo} - u_x^{abq})^2 + (u_y^{pogo} - u_y^{abq})^2}$ where u_x and u_y are the x and y displacements, as plotted in Fig. 8, and the superscripts pogo and abq indicate the Pogo and Abaqus solvers respectively.

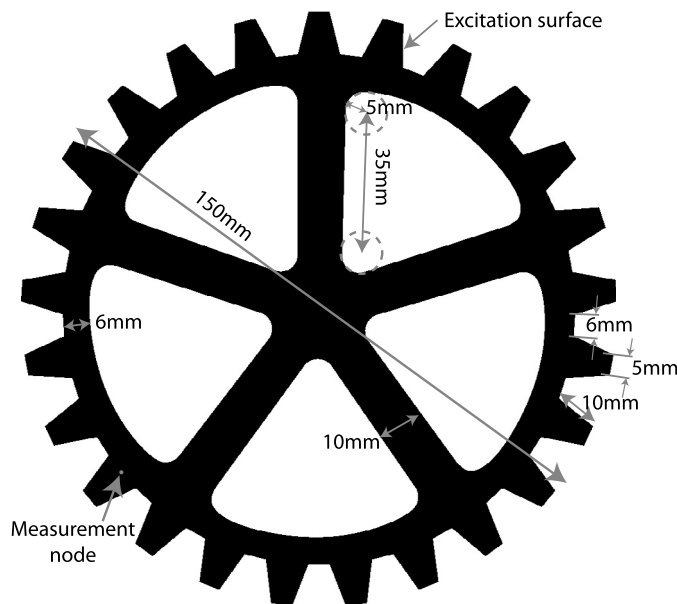


Fig. 10. The finite element model of the mechanical component.

The model was run using both Pogo and Abaqus, as before. Fig. 11 compares the displacements from the Abaqus model and the Pogo model. As before, by simply observing the time traces in Fig. 11(a), there is no evidence of any difference between the traces. Fig. 11(b) plots the discrepancy between the two, calculated in the same manner as for the weld model. This is fairly high, up to 4×10^{-13} , compared to the signals themselves which are around 2×10^{-11} ; this suggests a ‘signal to noise’ ratio of around 50, or 34 dB.

Fig. 11(c) plots the discrepancy over the entire time trace, and it becomes apparent that this increases over time. This is expected for long simulations; tiny discrepancies between the stiffness coefficients calculated between degrees of freedom are likely to cause increasingly large differences at each time step for explicit time domain simulations, causing the solutions to drift apart. In this case, the 500 000 time increments used is large compared to typical simulations (at least an order of magnitude larger than the other cases investigated in this paper), yet even at the end of the simulation the time traces

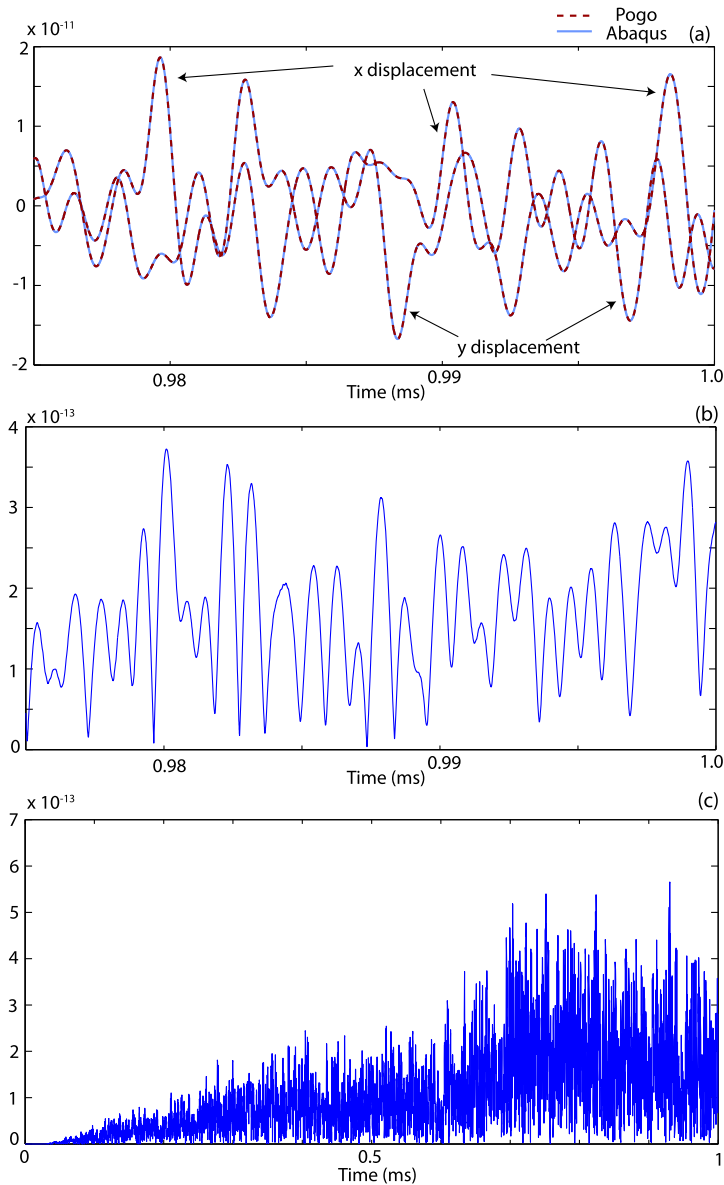


Fig. 11. Time trace comparison between Pogo and Abaqus for the mechanical component example. (a) shows a comparison between the two traces at the end of the simulation, and (b) plots the discrepancy between the two sets of data, as defined for Fig. 9. (c) plots the discrepancy between the two software packages over the entire simulation.

match reasonably well; it is unlikely that having potential errors at this level will cause significant problems for the majority of applications.

4.3. Geophysical example

The final example used to demonstrate Pogo is presented in Fig. 12. This is a representative geophysical example; the material properties are given in Table 3. The model is excited in direction (1,4) with a Gaussian of the form $f(t) = \exp[-1000(t - 0.25)^2]$. Absorbing regions are placed around three of the four sides; these are intended to minimise boundary reflections by using mass proportional damping. Such an approach is discussed in detail in [49]. Here, the damping coefficient is defined for each element as $d = 500(x/L)^3$ where x/L is the fraction of the total depth into the absorbing region. The absorbing regions were added to the model by a C++ program operating on the Abaqus input file.

A comparison of the time traces produced is presented in Fig. 13. In this example the traces are shown to match well, and the discrepancy between the time traces for the two models is around three orders of magnitude less than the displacements.

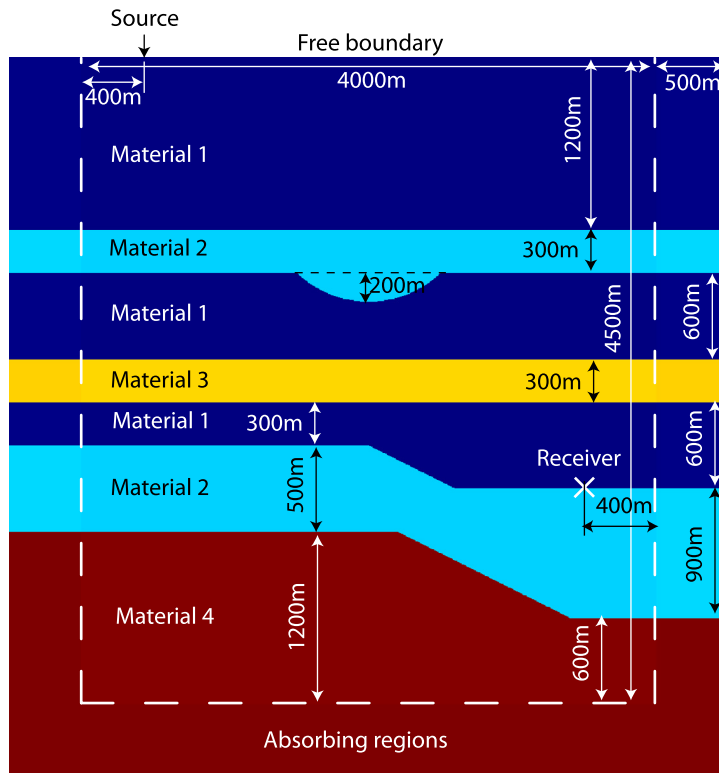


Fig. 12. The finite element geophysics model. The material properties are given in Table 3.

Table 3

Material properties of the different rock types for the geophysical model as shown in Fig. 12.

Material	Density (kg/m^3)	P-wave velocity, α (m/s)	S-wave velocity, β (m/s)
1	2600	4000	2000
2	2700	6000	3500
3	2500	3600	2100
4	2650	5000	2500

4.4. Discussion

Pogo has been shown to produce accurate results much faster than an established commercial CPU finite element software, Abaqus; in all cases Pogo is at least two orders of magnitude faster in total solution time. This observation must be put into perspective, however.

To analyse this, one can check the performance of the GPU kernel relative to the GPU capabilities itself. The algorithm is very memory intensive, i.e. very few calculations are actually performed for the amount of data loaded. This indicates that the algorithm is likely to be bandwidth limited, rather than compute limited (which is backed up by the use of the CUDA profiler available from Nvidia) and analysis of the performance should therefore focus on the bandwidth achieved.

Memory bandwidth for the algorithm is calculated as the total memory read from or saved to global memory per second. All the memory transactions carried out by the kernel are listed in Appendix A. This enables an estimate of the memory accessed per block, per time increment, to be determined; these values are listed in Table 4. The bandwidth can then be calculated by multiplying this by the number of blocks and by the number of time increments, then dividing by the execution time for the explicit time-stepping stage of the algorithm. Full details are in Appendix A. Table 4 presents the bandwidth for the three problems, in single and double precision, for the case with the aligned partitioner. The GTX 580 graphics card has a quoted bandwidth of 192.4 GB/s, and it is quite clear that the bandwidth achieved is around (and even beyond in a couple of cases) this limit. The kernel is therefore making good use of all the available bandwidth. It is noted in two cases that the quoted bandwidth exceeds the theoretical device peak by up to 2.5%. This is likely to be caused by inaccuracies in measuring the time and the various assumptions made in Appendix A when calculating the total memory accessed by the kernel. Additionally, there is a possibility that speed is improved by L2 caching, enabling previously accessed data to be re-used. Of the components listed in Table A.5 the only component which is accessed by multiple blocks is ' U_{curr} (linked)'; if this was fully cached it would reduce the global memory access by a block by around 2.5%, which could explain

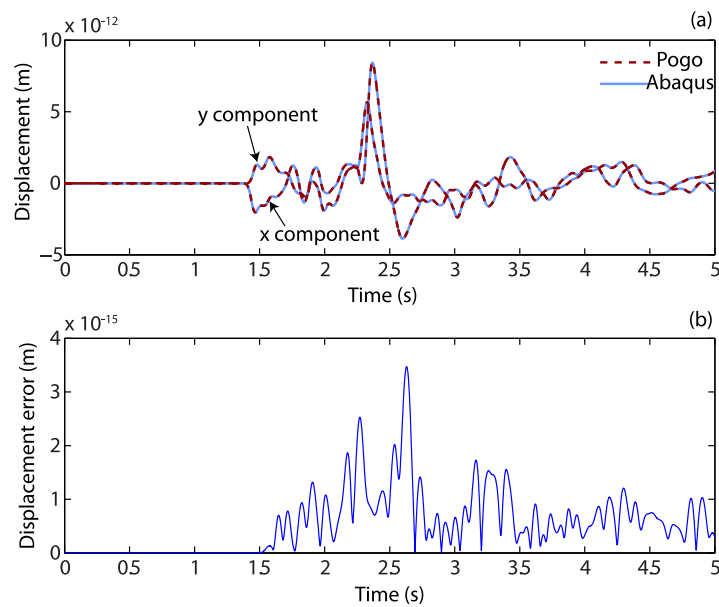


Fig. 13. Time traces for the geophysics model. (a) compares directly the Pogo and Abaqus results in the x and y directions at the receiver point marked in Fig. 12. (b) plots the discrepancy between the curves, as defined for Fig. 9.

Table 4

Bandwidth calculations. These are done for the aligned blocker, for single and double precisions calculated as discussed in Appendix A.

Model	Precision	n_{blocks}	Memory accessed per block (bytes)	Bandwidth (GB/s)
Weld	Single	1974	102 121	184
	Double	1974	188 803	197
Gear	Single	2931	103 080	184
	Double	2931	190 720	189
Geo.	Single	1979	105 587	191
	Double	1979	195 734	196

the small discrepancy. None the less these uncertainties are relatively small and it is clear that the algorithm is close to extracting the peak performance of the GPU.

We now return to the run-time comparison. Since we have established that the algorithm is essentially bandwidth limited, it can be assumed that an optimised CPU version would come close to maximising the memory bandwidth of the architecture. The Nehalem-based 5500 Xeon processor should be able to achieve a bandwidth of 12 GB/s; if this was fully utilised by the CPU version then the GPU would enable a speedup of around 16, rather than the 100–200 achieved compared to Abaqus.

There are a couple of points to note here. Firstly, as mentioned before, the GPU is a generation newer, so it could be expected that a factor of around 10 between the bandwidths would be more representative. Secondly, this demonstrates the inefficiency of Abaqus when simulating problems of this type. Clearly Abaqus in these circumstances is around an order of magnitude slower than the theoretical bandwidth limit would allow.

The method used to subdivide the mesh into separate blocks has a clear influence on Pogo's run time. In all cases, as expected, a better blocking efficiency leads to a shorter solution time. Since the efficiency is inversely proportional to the number of blocks, it might be expected that the run time would be inversely proportional to the efficiency. However, this is not the case; Pogo appears to perform better than expected with the underfilled blocks. The reason for this is that the Pogo algorithm does not specify that any calculations should be performed when encountering an unused node in a block; therefore if all the threads being run concurrently happen to be for unused nodes, CUDA can simply skip the calculations for all threads. This results in a slight speed improvement, although this is clearly less efficient than filling up all the blocks. A secondary important point relates to memory usage. All examples here were able to fit into the 3 GB of memory on the GTX 580 without any problem. However, for very large models where memory is limited, an efficient use of blocks may enable problems to be solved which would not otherwise be possible.

It is clear that the greedy blocker is significantly faster than the aligned blocker, typically by a factor of around 10. In two out of the three cases, the total simulation time is less when using the greedy blocker, although the actual solution stage is always faster using the more efficient aligned partitioning blocker.

Ideally, we would like to use the aligned partitioning to reduce solution time without the drawback of the additional pre-processing time. The codes for both blockers have been optimised to an extent, so it is unlikely that dramatic speed improvements could be achieved through this area. One important use of Pogo is to speed up parametric studies, where very similar jobs need to be run. Clearly if the mesh topology remains unchanged (for example, if only the location or nature of the forcing term changes, or if just the material properties change) then the same block file can be reused, so the run time of the blocker effectively becomes zero for all but the initial simulation. Alternatively, if there are only small topological changes, it may be possible to re-use the majority of the pre-existing block file, although an algorithm to perform this has not been developed yet.

This work has focused on taking an existing mesh and solving this problem on the GPU. It is important to recognise that the mesh must itself be generated initially, and this can be a slow process. For the mechanical component in Section 4.2, the mesh took a time of 12 minutes to produce using Abaqus's built in triangular mesher. Because of Pogo greatly speeding up the solution stage, pre-processing activities such as mesh generation can take up a disproportionate fraction of the total solution time; in this case the solution itself only takes 29 minutes, making meshing take 30% of the total solution time.

It is important to consider the precision of the solution approach. Double precision (i.e. using 64-bit floating point numbers) is often preferred for explicit time domain schemes, since the repeated time increments can lead to rounding errors with single precision which corrupt the results. While single precision times are included along with the double precision times for Pogo in Table 2, the main comparison between the two packages is presented using double precision data. Typically, double precision calculations perform poorly on 'gaming' GPUs such as the Geforce GTX 580 used in this study; this is because there is only one double precision calculation core for every six single precision cores. It would therefore appear surprising that the run-times reported in Table 2 approximately halve when using single rather than double precision; the run-times would be expected to reduce far more.

This is related to the fact that the algorithm is memory bandwidth limited rather than compute limited. It is well known that the use of linear finite elements with explicit time steps results in relatively few calculations being performed for a given amount of data being loaded from the memory; this is why it has been proposed to use higher order elements on GPUs [35] to increase the number of calculations per node at each time step. In the case of bandwidth limited algorithms such as this, since the amount of data which must be loaded from memory doubles for double precision, the run time will be expected to approximately double.

This study has focused on the use of the lowest order, linear triangular elements with explicit time steps because this method is widely used on the CPU and well understood. One option is to increase the order to quadratic elements, which can improve the accuracy by better representing the elastic field, with the disadvantage of increasing the computational complexity. The additional computational requirements are likely to have a low impact on speed for bandwidth limited algorithms, suggesting more accurate results could be obtained without significant additional execution time. However, a study of this is beyond the scope of this paper. Higher order elements have been used in the literature, but these are typically highly specialised, so the tools needed to, for example, generate meshes, are not widely available.

It is possible to extend the approach to 4-noded, linear quadrilateral elements. To do this, the non-zeros of the stiffness matrix K' should be treated just as in the 3-noded case: there will be a non-zero between any two nodes linked by an element. These non-zero 'links' can be identified at the pre-processing stage of the partitioning algorithm, then the partitioning and memory arrangement stages can treat these in the same way as if they had been produced from the triangular elements.

5. Conclusions

This paper has developed a pair of solutions to arrange an arbitrary mesh into memory to allow explicit time domain simulations to be performed on the GPU. One solution uses the well known 'greedy' domain partitioner to generate an initial subdivision of the mesh, and the second uses a more sophisticated 'aligned' partitioner which aims to arrange the subdivisions in a more optimal manner. The second stage of each takes the subdivided mesh and arranges the nodes within the GPU's memory to enable the separate partitions to communicate efficiently.

The memory arrangement can be passed to a solver, and a set of three of typical problems from non-destructive testing, engineering and geophysics were shown to run 100–200 times faster than that of a commercial CPU equivalent, Abaqus. The GPU algorithm was shown to use close to the maximum available bandwidth, but an equivalent optimised CPU version would be only 16 times slower, based on the available bandwidth, suggesting Abaqus is not optimal.

The software used to subdivide the mesh and the solver itself are all part of the package Pogo, which can be freely downloaded from <http://www.pogo-fea.com/>. Pogo is open source software, enabling members of the community to use, inspect and understand the code, and implement changes themselves. Also available as part of the package is a program which directly generates Pogo input files from Abaqus input files, and functions which enable the output files from Pogo to be loaded into Matlab.

Acknowledgement

The author is grateful to Professor M.J.S. Lowe for the critical reading of the paper and the helpful discussions.

Table A.5

Main memory transactions the Pogo kernel, for the 32×16 block; this is for double precision. Memory quantities are given in bytes. 'Block links' stores which adjacent block, and where within that block to load from. 'Source pointers' defines which of the time traces should be used as a source for each degree of freedom. 'Link pointers' specifies which of the data values loaded into memory should be multiplied by each K' term. L_{av} defines the average number of K' terms which must be loaded for each degree of freedom; $L_{av} = 9$ has been used here.

Description	Memory type	Size	Quantity	Total
U_{prev}	double	8	32×16	8192
U_{curr}	double	8	32×16	8192
Block links	int	4	20	80
U_{curr} (linked)	double	8	$32 \times 2 + 16 \times 10 + 8 \times 8$	4608
C'	double	8	32×16	4096
Source pointers	short int	2	$2 \times 32 \times 16$	2048
Fixed boundary condition	short int	2	32×16	1024
Link pointers	short int	2	$12 \times 32 \times 16$	12 288
K'	double	8	$4 \times 32 \times 16 \times L_{av}$	147 456
U_{next}	double	8	32×16	8192

Appendix A. Memory bandwidth calculation

To calculate the bandwidth achieved by Pogo, the total amount of global memory loaded and stored per block per time increment must first be evaluated. This is done by inspecting the kernel code to see what memory transactions are performed. Table A.5 lists these loads. Some additional memory transactions are not included, e.g. accessing the actual source time traces pointed to by the source pointers; these are typically likely to only be done for a few nodes in a model, so can be neglected; they are also difficult to quantify for a general case.

There is one parameter which must be estimated, L_{av} . Within the kernel, a 'for' loop exists, which loops through the maximum $L = 12$ links. In theory, if every block was entirely populated, and if every node had the maximum of 12 links to neighbours, L_{av} would be 12. An 'if' statement is used by the kernel within the for loop to check whether a K' term needs to be loaded. This means that if none of the threads within the warp require a term from K' , the load can be skipped. However, if just some of them do, the remaining threads must wait until the data has been loaded.

To calculate L_{av} , therefore, each block is split into its warps (these are 32 threads long, so this means each row is taken separately). The maximum number of links (i.e. non-zero K' terms) required by the nodes within each warp is taken, then L_{av} will correspond to the average of this across all the warps in all blocks.

This has been done assuming 100% blocking efficiency, so all threads within the block are used. If this is not 100%, the end rows in the block can be left without any threads at all. For these rows, the number of K' loads which need to be performed is 0.

For our purposes, L_{av} is estimated. It is assumed that the average maximum links per warp, assuming 100% efficient blocking is 9, which is based on experience with several typical free mesh models. The average is then reduced by multiplying by the block efficiency defined in the paper to account for the warps of unused nodes.

Based on Table A.5, the total memory loaded and saved per block is 196 176 bytes. Of this, 75% corresponds to loading the K' coefficients, indicating how critical this point is to the execution time of the algorithm. When the block efficiency is not 100%, the memory can be estimated as $48 720 + 147 456 \times \eta$. The bandwidth used is calculated by multiplying this number by the number of blocks, then by the number of increments, then dividing by the amount of time taken for the explicit stage of the algorithm.

References

- [1] NVIDIA, CUDA C Programming Guide, <http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>.
- [2] J.A. Anderson, C.D. Lorenz, A. Travasset, General purpose molecular dynamics simulations fully implemented on graphics processing units, J. Comput. Phys. 227 (10) (2008) 5342–5359, <http://dx.doi.org/10.1016/j.jcp.2008.01.047>.
- [3] G.R. Joldes, A. Wittek, K. Miller, Real-time nonlinear finite element computations on GPU – Application to neurosurgical simulation, Comput. Methods Appl. Mech. Eng. 199 (49) (2010) 3305–3314, <http://dx.doi.org/10.1016/j.cma.2010.06.037>.
- [4] C. Cecka, A.J. Lew, E. Darve, Assembly of finite element methods on graphics processors, Int. J. Numer. Methods Eng. 85 (5) (2011) 640–669, <http://dx.doi.org/10.1002/nme.2989>.
- [5] J. Lengyel, M. Reichert, B.R. Donald, D.P. Greenberg, Real-time robot motion planning using rasterizing computer graphics hardware, SIGGRAPH Comput. Graph. 24 (4) (1990) 327–335, <http://dx.doi.org/10.1145/97880.97915>.
- [6] K.E. Hoff, J. Keyser, M. Lin, D. Manocha, T. Culver, Fast computation of generalized Voronoi diagrams using graphics hardware, in: Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '99, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1999, pp. 277–286, <http://dx.doi.org/10.1145/311535.311567>.
- [7] M. Woo, J. Neider, T. Davis, D. Shreiner, OpenGL programming guide: the official guide to learning OpenGL, version 1.2, Addison-Wesley Longman Publishing Co., Inc., 1999.
- [8] C. Trendall, J. Stewart, General calculations using graphics hardware, with application to interactive caustics, in: B. Peroche, H. Rushmeier (Eds.), Rendering Techniques '00, Eurographics, Springer-Verlag, Wien, New York, Brno, Tchéque République, 2000, <http://hal.inria.fr/inria-00510058>.
- [9] M.J. Harris, G. Coombe, T. Scheuermann, A. Lastra, Physically-based visual simulation on graphics hardware, in: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware, HWWs '02, Eurographics Association, Aire-la-Ville, Switzerland, 2002, pp. 109–118, <http://dl.acm.org/citation.cfm?id=569046.569061>.

- [10] M. Macedonia, The GPU enters computing's mainstream, *Computer* 36 (10) (2003) 106–108, <http://dx.doi.org/10.1109/MC.2003.1236476>.
- [11] J. Krüger, R. Westermann, Linear algebra operators for GPU implementation of numerical algorithms, *ACM Trans. Graph.* 22 (3) (2003) 908–916, <http://dx.doi.org/10.1145/882262.882363>.
- [12] J. Bolz, I. Farmer, E. Grinspun, P. Schröder, Sparse matrix solvers on the GPU: conjugate gradients and multigrid, *ACM Trans. Graph.* 22 (3) (2003) 917–924, <http://dx.doi.org/10.1145/882262.882364>.
- [13] C.J. Thompson, S. Hahn, M. Oskin, Using modern graphics architectures for general-purpose computing: a framework and analysis, in: *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture, MICRO 35*, IEEE Computer Society Press, Los Alamitos, CA, USA, 2002, pp. 306–317, <http://dl.acm.org/citation.cfm?id=774861.774894>.
- [14] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, P. Hanrahan, Brook for GPUs: stream computing on graphics hardware, *ACM Trans. Graph.* 23 (3) (2004) 777–786, <http://dx.doi.org/10.1145/1015706.1015800>.
- [15] A.M. Bayoumi, M. Chu, Y.Y. Hanafy, P. Harrell, G. Refai-Ahmed, Scientific and engineering computing using ATI stream technology, *Comput. Sci. Eng.* 11 (6) (2009) 92–97.
- [16] A. Munshi, The OpenCL Specification v1.2, <http://www.khronos.org/opencl/>, November 2012.
- [17] A. Balevic, L. Rockstroh, A. Tausendfreund, S. Patzelt, G. Goch, S. Simon, Accelerating simulations of light scattering based on finite-difference time-domain method with general purpose GPUs, in: *11th IEEE International Conference on Computational Science and Engineering*, IEEE, 2008, pp. 327–334, <http://dx.doi.org/10.1109/CSE.2008.16>.
- [18] D. De Donno, A. Esposito, L. Tarricone, L. Catarinucci, Introduction to GPU computing and CUDA programming: A case study on FDTD [EM programmer's notebook], *IEEE Antennas Propag. Mag.* 52 (3) (2010) 116–122, <http://dx.doi.org/10.1109/MAP.2010.5586593>.
- [19] D. Michéa, D. Komatitsch, Accelerating a three-dimensional finite-difference wave propagation code using GPU graphics cards, *Geophys. J. Int.* 182 (1) (2010) 389–402, <http://dx.doi.org/10.1111/j.1365-246X.2010.04616.x>.
- [20] A.C. Cangellaris, D.B. Wright, Analysis of the numerical error caused by the stair-stepped approximation of a conducting boundary in FDTD simulations of electromagnetic phenomena, *IEEE Trans. Antennas Propag.* 39 (10) (1991) 1518–1525, <http://dx.doi.org/10.1109/8.97384>.
- [21] M. Drozd, Efficient finite element modelling of ultrasound waves in elastic media, Ph.D. thesis, Imperial College London, 2008.
- [22] P. Huthwaite, F. Simonetti, M.J.S. Lowe, On the convergence of finite element scattering simulations, in: *AIP Conference Proceedings*, vol. 1211, 2010, p. 65.
- [23] Z. Fan, F. Qiu, A. Kaufman, S. Yoakum-Stover, GPU cluster for high performance computing, in: *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, IEEE Computer Society, Washington, DC, USA, 2004, p. 47, <http://dx.doi.org/10.1109/SC.2004.26>.
- [24] K. Liu, X.-B. Wang, Y. Zhang, C. Liao, Acceleration of time-domain finite element method (TD-FEM) using graphics processor units (GPU), in: *7th International Symposium on Antennas, Propagation EM Theory*, 2006, pp. 1–4, <http://dx.doi.org/10.1109/ISAPE.2006.353223>.
- [25] D. Göddeke, R. Strzodka, S. Turek, Accelerating double precision FEM simulations with GPUs, in: *Proceedings of ASIM – 18th Symposium on Simulation Technique*, 2005.
- [26] D. Göddeke, R. Strzodka, J. Mohd-Yusof, P. McCormick, S.H.M. Buijssen, M. Grajewski, S. Turek, Exploring weak scalability for FEM calculations on a GPU-enhanced cluster, *Parallel Comput.* 33 (10–11) (2007) 685–699, <http://dx.doi.org/10.1016/j.parco.2007.09.002>.
- [27] D. Göddeke, R. Strzodka, S. Turek, Performance and accuracy of hardware-oriented native-, emulated- and mixed-precision solvers in FEM simulations, *Int. J. Parallel Emerg. Distrib. Syst.* 22 (4) (2007) 221–256, <http://dx.doi.org/10.1080/17445760601122076>.
- [28] D. Göddeke, Fast and accurate finite-element multigrid solvers for PDE simulations on GPU clusters, Ph.D. thesis, Technische Universität Dortmund, Fakultät für Mathematik, May 2010, <http://hdl.handle.net/2003/27243>, <http://www.logos-verlag.de/cgi-bin/buch?isbn=2768>.
- [29] S. Turek, D. Göddeke, S.H.M. Buijssen, H. Wobker, Hardware-oriented multigrid finite element solvers on GPU-accelerated clusters, in: J. Kurzak, D.A. Bader, J.J. Dongarra (Eds.), *Scientific Computing with Multicore and Accelerators*, CRC Press, 2010, Ch. 6.
- [30] M. Geveler, D. Ribbrock, D. Göddeke, P. Zajac, S. Turek, Towards a complete FEM-based simulation toolkit on GPUs: Unstructured grid finite element geometric multigrid solvers with strong smoothers based on sparse approximate inverses, *Comput. Fluids* 80 (2013) 327–332, <http://dx.doi.org/10.1016/j.compfluid.2012.01.025>.
- [31] D. Göddeke, C. Becker, S. Turek, Integrating GPUs as fast co-processors into the parallel FE package FEAST, in: M. Becker, H. Szczerbicka (Eds.), *19th Symposium Simulationstechnique (ASIM'06)*, *Frontiers in Simulation*, 2006, pp. 277–282.
- [32] O. Comas, Z.A. Taylor, J. Allard, S. Ourselin, S. Cotin, J. Passenger, Efficient nonlinear FEM for soft tissue modelling and its GPU implementation within the open source framework SOFA, in: F. Bello, P. Edwards (Eds.), *Biomedical Simulation*, in: *Lecture Notes in Computer Science*, vol. 5104, Springer, Berlin, Heidelberg, 2008, pp. 28–39.
- [33] C. Dick, J. Georgii, R. Westermann, A real-time multigrid finite hexahedra method for elasticity simulation using CUDA, *Simul. Model. Pract. Theory* 19 (2) (2011) 801–816, <http://dx.doi.org/10.1016/j.simpat.2010.11.005>.
- [34] D. Komatitsch, D. Michéa, G. Erlebacher, Porting a high-order finite-element earthquake modeling application to NVIDIA graphics cards using CUDA, *J. Parallel Distrib. Comput.* 69 (5) (2009) 451–460.
- [35] A. Klöckner, T. Warburton, J. Bridge, J.S. Hesthaven, Nodal discontinuous Galerkin methods on graphics processors, *J. Comput. Phys.* 228 (21) (2009) 7863–7882, <http://dx.doi.org/10.1016/j.jcp.2009.06.041>.
- [36] S.H. Hsieh, G.H. Paulino, J.F. Abel, Evaluation of automatic domain partitioning algorithms for parallel finite element analysis, *Int. J. Numer. Methods Eng.* 40 (6) (1997) 1025–1051, [http://dx.doi.org/10.1002/\(SICI\)1097-0207\(19970330\)40:6<1025::AID-NME103>3.0.CO;2-P](http://dx.doi.org/10.1002/(SICI)1097-0207(19970330)40:6<1025::AID-NME103>3.0.CO;2-P).
- [37] W. Lord, R. Ludwig, Z. You, Developments in ultrasonic modeling with finite element analysis, *J. Nondestruct. Eval.* 9 (2) (1990) 129–143, <http://dx.doi.org/10.1007/BF00566389>.
- [38] Y. Lin, M. Sansalone, N.J. Carino, Finite element studies of the impact-echo response of plates containing thin layers and voids, *J. Nondestruct. Eval.* 9 (1) (1990) 27–47, <http://dx.doi.org/10.1007/BF00566980>.
- [39] F. Moser, L.J. Jacobs, J. Qu, Modeling elastic wave propagation in waveguides with the finite element method, *Nondestruct. Test. Eval. Int.* 32 (4) (1999) 225–234, [http://dx.doi.org/10.1016/S0963-8695\(98\)00045-0](http://dx.doi.org/10.1016/S0963-8695(98)00045-0).
- [40] G. Baskaran, C.L. Rao, K. Balasubramaniam, Simulation of the TOFD technique using the finite element method, *Insight* 49 (11) (2007) 641–646, <http://dx.doi.org/10.1784/insi.2007.49.11.641>.
- [41] P. Moczo, J. Kristek, M. Galis, P. Pazak, M. Balazovjeh, The finite-difference and finite-element modeling of seismic wave propagation and earthquake motion, *Acta Phys. Slovaca* 57 (2) (2007) 177–406, <http://dx.doi.org/10.2478/v10155-010-0084-x>.
- [42] R. Cook, et al., *Concepts and Applications of Finite Element Analysis*, John Wiley & Sons, 2007.
- [43] K.J. Bathe, E.L. Wilson, *Numerical Methods in Finite Element Analysis*, Prentice-Hall, Englewood Cliffs, NJ, 1976.
- [44] E. Cuthill, J. McKee, Reducing the bandwidth of sparse symmetric matrices, in: *Proceedings of the 1969 24th National Conference*, ACM, 1969, pp. 157–172, <http://dx.doi.org/10.1145/800195.805928>.
- [45] J.C. Caendish, D.A. Field, W.H. Frey, An approach to automatic three-dimensional finite element mesh generation, *Int. J. Numer. Methods Eng.* 21 (2) (1985) 329–347, <http://dx.doi.org/10.1002/nme.1620210210>.
- [46] H. Jin, N.E. Wiberg, Two-dimensional mesh generation, adaptive remeshing and refinement, *Int. J. Numer. Methods Eng.* 29 (7) (1990) 1501–1526, <http://dx.doi.org/10.1002/nme.1620290709>.

- [47] P.L. George, E. Seveno, The advancing-front mesh generation method revisited, *Int. J. Numer. Methods Eng.* 37 (21) (1994) 3605–3619, <http://dx.doi.org/10.1002/nme.1620372103>.
- [48] Dassault Systèmes Simulia Corp., Abaqus 6.11 documentation, <http://abaqus.ethz.ch:2080/v6.11/>.
- [49] P. Rajagopal, M. Drozd, E. Skelton, M.J.S. Lowe, R. Craster, On the use of absorbing layers to simulate the propagation of elastic waves in unbounded isotropic media using commercially available finite element packages, *NDT & E International*, <http://dx.doi.org/10.1016/j.ndteint.2012.04.001>.