Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 1 (2012) 1093–1100

Procedia Computer Science

www.elsevier.com/locate/procedia

International Conference on Computational Science, ICCS 2010

3D finite element numerical integration on GPUs

Paweł Maciołᵃ, Przemysław Płaszewskiᵇ, Krzysztof Banaśᵇ,ᵃ

ᵃ*Institute of Computer Modelling,*
Cracow University of Technology, Warszawska 24, 31-155 Kraków, Poland
ᵇ*Department of Applied Computer Science and Modelling,*
AGH University of Science and Technology, Mickiewicza 30, 30-059 Kraków, Poland

Abstract

The algorithmic and computational aspects of 3D finite element numerical integration on GPUs are investigated in the paper. The special stress is put on selecting the proper parallelization strategies depending upon the properties of FEM problems solved and approximations used. The close interplay between the available computational resources of GPUs and the possible implementation strategies and obtained results is observed.

Keywords: GPGPU, numerical integration, finite elements, parallel programming, higher order approximation

1. Finite element numerical integration

Numerical integration is the process that leads from a weak, finite element statement of the problem solved to the system of linear equations, solution of which produces a finite element approximation. The standard procedure consist of a loop over finite elements into which the computational domain is discretized. For each element a, so called, element stiffness matrix is produced and than assembled into the global matrix – the matrix of a solved system of linear equations, called also the global stiffness matrix.

The level of complexity of creation of the element stiffness matrix depends on the problem solved and the approximation used. In theory, each entry of the element stiffness matrix is obtained by evaluation of integrals comprising the weak, finite element statement of the problem. In practice, for many simple elements and approximations, integrals can be precomputed analytically and creation of stiffness matrix entries changes into algebraic operations, with closed formulae into which element parameters, like e.g. vertices coordinates, are substituted. In such cases creation of the global stiffness matrix consist mainly of assembling of quickly calculated element stiffness matrices [1].

There are, however, situations, e.g. for nonlinear problems (with PDE coefficients depending on the current solution) or for complex elements with curvilinear shapes, where integrals cannot be precomputed. In these cases numerical integration has to be performed and can form a substantial, in terms of required resources, part of the finite element solution process, especially for higher order approximation. The current article investigates such cases, focusing on integration procedure, without considering the further assembly step.

Email address: kbanas@pk.edu.pl (Krzysztof Banaś)

1877-0509
doi:10.1016/j.procs.2010.04.121

	Degree of approximation p						
Number of:	1	2	3	4	5	6	7
shape functions	6	18	40	75	126	196	288
$\boldsymbol{A}^e$ entries	36	324	1600	5625	15876	38416	82944
Gaussian points	6	18	48	80	150	231	336

Table 1: Parameters determining the computational characteristics of finite element numerical integration – the number of shape functions, the number of element stiffness matrix entries and the number of Gaussian integration points – as functions of the degree of approximating polynomials

2. The integration algorithms

As a model problem we consider simple Laplace's equation in a 3D domain. In computing element stiffness matrix entries, the most important, from the performance point of view, are calculations of the domain integral:

$$\int_{\Omega_e} \frac{\partial \hat{\phi}^r}{\partial x_i} \frac{\partial \hat{\phi}^s}{\partial x_i} d\Omega \tag{1}$$

where Ω_e is the domain of an element, $\hat{\phi}^r$ are element shape functions (forming a basis for approximation) and the summation convention for repeated indices is used. Each integral (1) corresponds to a pair of shape functions $\hat{\phi}^r$ and $\hat{\phi}^s$, $r, s = 1, .., N_{\text{sh}}$, and contributes to a single entry $A^e_{r,s}$ of the element stiffness matrix $\boldsymbol{A}^e$. Hence in each element we compute many integrals, each of which corresponds to a single entry in $A^e_{r,s}$, i.e. a pair of shape functions.

The number of entries calculated in each element depends on the number of shape functions, N_{sh}. The number of shape functions depend on the order of approximation. For higher orders of approximation we use rich finite element spaces with many shape functions (some of them being polynomials of higher degree). For low order approximations (e.g. linear) we use basic finite element spaces, most often with linear or multi-linear shape functions. For our model problem we use prismatic elements and shape functions being products of 2D shape functions for triangles and 1D shape functions in vertical direction. Table 1 gives the number of shape functions for this combination, as a function of the degree of approximating polynomial, p.

Numerical integration in finite element codes is usually done in two steps. First, the change of the domain of integration, from a real element (with $\boldsymbol{x}$ coordinates) to a reference element (with $\boldsymbol{\xi}$ coordinates) is performed. Then standard numerical integration is applied. There are only several reference elements used in finite element codes and for each of them there are prescribed quadratures allowing for sufficiently accurate integration of finite elements integrals. The most common are Gauss-Legendre quadratures that we also employ in our investigations.

Finite element Gaussian integration transforms integral (1) into a sum over integration points $\boldsymbol{\xi}_I$ with weights w_I, $I = 1, .., N_I$:

$$\sum_{I=1}^{N_I} \left(\frac{\partial \hat{\phi}^r}{\partial \boldsymbol{\xi}} \frac{\partial \boldsymbol{\xi}}{\partial x_i} \right) \left(\frac{\partial \hat{\phi}^s}{\partial \boldsymbol{\xi}} \frac{\partial \boldsymbol{\xi}}{\partial x_i} \right) \det \boldsymbol{J}_{\boldsymbol{T}_e} w_I \tag{2}$$

All values in (2) are computed at integration points $\boldsymbol{\xi}_I$ and $\boldsymbol{J}_{\boldsymbol{T}_e}$ denotes the Jacobian matrix of transformation $\boldsymbol{T}_e$ from the reference element $\hat{\Omega}_e$ to the real element Ω_e. $\frac{\partial \boldsymbol{\xi}}{\partial x_i}$ forms a column of the Jacobian matrix $\boldsymbol{J}_{\boldsymbol{T}_e^{-1}}$, $\boldsymbol{J}_{\boldsymbol{T}_e^{-1}} = \frac{\partial \boldsymbol{\xi}}{\partial \boldsymbol{x}}$, that corresponds to the inverse transformation $\boldsymbol{T}_e^{-1}$.

The number of Gaussian points, N_I, is determined by the requirements of convergence of the finite element solution [2]. It also depends on the order of approximation and usually (with some departures for special kinds of problems) is chosen so as to integrate exactly finite element integrals in a reference element. Table 1 shows the number of Gaussian points used in our study.

3. Computational aspects of numerical integration

To create an element stiffness matrix, N^2_{sh} integrals (entries) have to be computed, each as a sum of contributions from different Gaussian points. Hence, in the numerical integration algorithm there are three loops: two over shape functions and one over integration points.

The size of: (in kB)	Degree of approximation p 1	2	3	4	5	6	7
$\hat{\phi}_i$, $\frac{\partial \hat{\phi}_i}{\partial \boldsymbol{x}}$	0.09	0.28	0.63	1.17	1.97	3.06	4.50
A^e entries	0.14	1.27	6.25	21.97	62.02	150.06	324.00
$\boldsymbol{\xi}^I$, w^I	0.09	0.28	0.75	1.25	2.34	3.61	5.25
$\hat{\phi}_i$, $\frac{\partial \hat{\phi}_i}{\partial \boldsymbol{x}}$ at all $\boldsymbol{\xi}^I$	0.56	5.06	30.00	93.75	295.31	707.44	1512.00

Table 2: The size (in kilobytes) of data used in numerical integration algorithm: values of shape functions and their derivatives, element stiffness matrix entries, Gaussian integration points and weights

It is possible to perform integration naively: for each pair of shape functions compute the corresponding integral in a loop over integration points. At each point, values of shape functions are calculated and used in summations. For the next pair of shape functions the procedure repeats. This however neglects the fact that computing the values of different shape functions at a specified point is strongly related. Instead of repeating calculations of values for each particular pair of shape functions the procedure is reversed.

The loop over integration points is the outer loop. At each integration point the values of all shape functions are calculated and contributions from the given integration point to all element stiffness matrix entries are added in a double loop over shape functions.

To design parallelization strategies for different hardware and software environments it is necessary to realize what are the resources required for performing necessary calculations.

There are data that need to be retrieved from finite element data structures. These include geometrical data for elements (usually in the form of vertices coordinates but for curvilinear elements more data may be necessary) and coefficients for the solved PDEs (or some parameters needed for evaluating coefficients). In the case of our model problem we assume that only coordinates of six element vertices are passed as arguments to the integration procedure.

The next important data are the coordinates of integration points and weights associated with them. For each reference element of a given type and given degree of approximation these data are the same. Hence it is advantageous to put these data in some fast memory, possibly read only, available to all working threads. Moreover, the loop over elements, the outermost loop of the global stiffness matrix creation, should be organized in such a way, that elements are grouped into sets of elements with the same type and degree of approximation p.

Hence in our study we investigate the case in which integration is done for a sequence of elements of the same type and degree of approximation. Performance data are obtained for such a case and does not take into account the situation in which elements of different type or approximation order are sent to CPU or GPU for processing.

When looking for the part of memory where integration data can be placed it is important to realize its required size. In terms of the number of scalars it can be quickly calculated using Table 1. In terms of kilobytes it is given in Table 2 (because we consider older types of GPUs we assume floating point data with single precision only).

The last group of data used in final calculations of approximate integrals are the values of shape functions and their derivatives. Once again these values are the same for all elements of a given type and degree of approximation. Hence they can be precomputed and stored in some quickly accessible memory. However, in such a case the memory requirements grow substantially. Table 2 presents in its first line only the size of $\hat{\phi}_i$ and $\frac{\partial \hat{\phi}_i}{\partial \boldsymbol{x}}$ for a single integration point. The last row of Table 2 shows the required size of data structures storing the values of shape functions and their derivatives for all integration points.

In a competitive strategy of computing the values of shape functions separately for each element, the values of $\hat{\phi}_i$ and $\frac{\partial \hat{\phi}_i}{\partial \boldsymbol{x}}$ are computed within the loop over integration points. At each point all values are computed but only at this point. The efficiency of the procedure depends on the kind of shape functions (whether they are standard Lagrangian, hierarchical, obtained as tensor products, etc.).

For lower orders of approximation the option of precomputing the values of shape functions and their derivatives may be attractive. The necessary storage remains small. When deciding which strategy to choose one has to take into account the characteristics of computing platforms – the time to perform necessary calculations and the time to retrieve precomputed values. The larger the size of data structures, the higher probability that they will not fit into a fast memory. For really high orders of approximations the required memory resources may not even be available.

The number of	Degree of approximation p								
operations $\times 10^{-3}$	1	2	3	4	5	6	7	8	9
outside loops	0.5	0.8	1.5	2.6	4.2	6.3	9.2	12.8	17.3
inside loops	0.2	2.2	11.2	39.3	111.1	268.9	580.6	1148.1	2117.5

Table 3: The number of operations (in thousands) performed at each integration point for the model problem, split into operations performed outside loops over shape functions and inside loops

Because of greater flexibility of this strategy and the known fact of limited resources available for GPU computations, we consider in our investigations only the case of computing shape functions separately for each element. Then, the required memory resources are moderate and do not vary significantly with p.

There is a different situation with the resulting element stiffness matrix. Its size changes (see Table 2) from less than one kilobyte to several hundreds kilobytes. For execution on large CPU cores with large caches this does not pose substantial problems, for other hardware architecture the chosen strategy for parallelization may be influenced by the requirements of properly storing the element stiffness matrix.

Another factor that has to be taken into account when designing parallelization strategies for numerical integration is the number of operations performed within subsequent loops of the algorithm. The general orders of magnitude results from the number of integration points and shape functions, however exact quantities depend on the problem solved, approximation used and the geometry of finite elements.

Table 3 presents these numbers for our model problem, split into two groups (only floating point operations are counted): operations performed outside loops over shape functions and operations performed within these loops.

The values in the table correspond to a single integration point and both groups are calculated assuming that shape functions are computed separately for each element and each integration point. In this particular case, the number of operations for a single pair of shape functions at a given integration point is equal to 7 (3 multiplications of derivatives with respect to three space dimensions, 3 summations and 1 additional multiplication by $\det \boldsymbol{J}_{\boldsymbol{T}_e} \cdot w_I$ – precomputed earlier). For other types of problems (especially nonlinear) this number may be much higher. However, usually this would mean that the number of operations per memory access will be greater and obtained parallel speed-ups should also increase.

The reason for the splitting presented in Table 3 is the fact that only operations within loops over shape functions are easily parallelizable. There are no dependencies between calculations for two separate entries of the element stiffness matrix. To the contrary, in calculations outside loops over shape functions there are many strong dependencies making parallelization difficult or even impossible. Both, computing the values of shape functions and their derivatives, as well as computing parameters depending upon the geometry of an element (Jacobian matrices, determinant, derivatives with respect to physical coordinates), especially for the most popular geometrically linear and multilinear elements, have small potential for parallelization.

4. Mapping of numerical integration algorithm to GPU architecture

The most important characteristics of GPU architectures are: massive parallelism and rich memory hierarchy with complex optimal access patterns. The potential for concurrency in finite element numerical integration is related mainly to two factors: the number of elements for which integration is performed and the number of entries for a single element stiffness matrix.

For standard CPU cores the first option is easier to exploit. Following the standard approach of domain decomposition the loop over all elements can be split and executed on as many cores as there are elements. One thread calculates stiffness matrices for several subsequent elements, making use of large caches to improve performance. Calculations for different elements are practically independent (different elements can use the same geometrical data for common parts but this will not have any impact on performance). The only dependencies appear in the assembly phase and they have to be properly resolved. Hence, finite element numerical integration is perfectly parallel for CPU cores – the time will decrease proportionally to the number of cores and the speed-up should approach the perfect line.

For GPU architectures there appear a problem of small resources available for a single thread. For a single element we should put the data structures from the first three lines and a particular column in Table 2 in fast memory available to the thread. Assuming, for example, that we want to use concurrently 32 threads (minimal number for CUDA programming model), we have to provide a fast memory with the size 32 times larger than the values in the table. While for $p = 1$ this means only approximately 10 kBytes, for p greater than 4 it reaches more than 1 MByte.

Therefore we consider another approach where the decomposition concerns the resulting element stiffness matrix. All entries are divided among threads and each thread computes the values of several entries. When adopting this parallelization strategy the efficiency would result from the proper placement of data structures in the available memories and the proper organization of calculations, that would ensure optimal access patterns to data structures.

We consider the implementation of numerical integration algorithm for CUDA execution environment [3]. From the point of view of organization of threads the environment is characterized by:

- grouping of threads into warps, sets of 32 threads simultaneously executed by hardware
- grouping of warps into threadblocks, sets of threads accessing the same shared memory and synchronized by fast operations
- grouping of threadblocks into a grid, a set of threads executing the same kernel

The available memory consist of:

- limited number of registers allocated by the compiler
- 16 kB of fast (shared) memory
- so called constant memory, filled by the code running on the host CPU and quickly accessible thanks to caching
- large device memory, relatively slow – both when used for transfers from and to the main memory of the host computer and when accessed by threads

Because of the reasons described previously the main assumption of the implementation is that a single finite element corresponds to a single threadblock and that individual threads calculate sets of element stiffness matrix entries. The whole grid of threadblocks correspond to the whole finite element mesh (or possibly a submesh for large problems or parallel execution in distributed environments).

The size of threadblocks is one of the key characteristics of the parallelization strategy. It is always a multiple of 32. From all the threads in the threadblock, only those for which there are entries in the stiffness matrix to be computed are active. The rest remains inactive – do not perform calculations.

The number of computed element stiffness matrix entries cannot be smaller than the number of active threads in a threadblock, however may be larger. In such a case there are several entries per thread. For large element stiffness matrices one can select different numbers of active threads (different sizes of threadblocks) with a different number of element stiffness matrix entries per thread.

The actual decision on the size of threadblocks depends on the amount of required memory resources. We try to optimize memory accesses by threads and, on the other hand, try to minimize the amount of memory used. On one hand we want the number of threads in a threadblock to be large (as is recommended for CUDA architecture, since it helps to efficiently schedule threads), but on the other hand we want to allow for more than one threadblock to perform computations on each multiprocessor (since this also increase the efficiency of computations).

To allow for optimal access by the threads, the data structures, when necessary, are padded, making their size a multiple of 32, according to the requirements of CUDA architecture. This concerns the input array with element data, as well as the output array with stiffness matrix entries.

The resulting element stiffness matrix for orders of approximation greater than 3 is too big to fit into the fast shared memory. For these cases computations are split, the array is divided into horizontal stripes of several rows, and subsequent stripes are calculated one after another, with threads always operating on shared memory. The penalty for this splitting is that for each stripe of the element stiffness matrix the values of shape functions and their derivatives are calculated again.

The algorithm proceeds in the following steps:

- element data (vertex coordinates, PDE coefficients, etc.) for all elements related to a given grid of threadblocks are placed in special, padded arrays and copied from the host computer memory to the CUDA device memory in one call
- Gaussian integration data are copied to constant memory
- a proper integration kernel function is invoked to be executed by concurrent threads
- output data (in the form of element stiffness matrices) are copied from the device memory to the host memory

A single kernel function performs integration for a set of element stiffness matrix entries in the following order:

- based on its ID the thread realizes which element and which stiffness matrix entries are assigned to it
- element data are copied from device to shared memory by first 32 threads in the block (the array with element data is padded so there are no bank conflicts when accessing memory)
- because of the possible large size of the element stiffness matrix the outermost loop is the loop over stripes of the matrix
- the next loop is the standard loop over integration points
- for each integration point (and each stripe!) the values of all shape functions and their derivatives are calculated by a single thread
- geometrical quantities (Jacobian matrices etc.) are computed by a single thread
- derivatives of shape functions with respect to physical coordinates are calculated in parallel by the respective number of threads
- in a double loop over the element stiffness matrix entries corresponding to a given thread the calculations of entries are performed
- the entries for a given stripe of the element stiffness matrix are copied from the shared to the device memory (with coalesced memory accesses)

5. Numerical experiments

The developed algorithm has been tested using an NVIDIA GeForce 8800GTX graphics card (with CUDA SDK 2.3 and Microsoft Visual Studio 2008 C/C++ compiler and environment). The performance results have been compared with those of a single core of AMD X2 processor (with 2.6GHz clock and 1MB L2 cache and the same C/C++ compiler and environment as GPU). Compilers were used with simple *-O2* optimization flag.

Both, the GPU and the CPU were performing single precision computations. The use of single precision was dictated by the capabilities of the GPU utilized. For newer GPUs double precision calculations are also available. The GFlops performance in this case is however worse. This can also concern CPU cores – vector instructions for single precision numbers can be twice as fast as those for the double precision. Whether to use single or double precision becomes a computational as well as numerical problem. Properly optimized single precision calculations can be faster, but the accuracy of the final finite element solution may require the use of double precision numbers (or special mixed precision refinement [4]). The final decision which type of computations to apply will eventually depend on the finite element problem solved and such factors as e.g. stability of the FEM algorithm or the condition number of the obtained system of linear equations [5, 6].

Table 4 presents execution parameters that characterize the utilization of GPU resources during numerical integration. It can be seen how a proper number of stiffness matrix (SM) entries has to be chosen to achieve a balance between the degree of concurrency within a single threadblock (more SM entries per thread means less redundant shape functions calculations) and the concurrency among different threadblocks (less SM entries per thread means less GPU resources per threadblock and more active blocks per GPU multiprocessors).

	p=1	p=2	p=3	p=4	p=5	p=6	p=7
The number of threads in a block	64	128	128	384	128	256	288
The number of working threads	36	108	128	375	126	196	288
The number of SM entries per thread	1	3	12.5	15	126	196	288
The number of SM rows proceeded at once	6 (all)	18 (all)	40 (all)	25	6	7	6
The size of padded stiffness matrix	64	384	1600	5625	15876	38416	82944
The number of registers per thread	16	18	22	20	21	21	24
Shared memory per block (in kB)	0.84	5.51	6.90	8.84	5.04	9.96	10.91
The number of active blocks	8	2	2	1	2	1	1

Table 4: The characteristics of the execution of the numerical integration algorithm by NVIDIA GeForce 8800GTX graphics card for the model problem and different orders of approximation p

	p=1	p=2	p=3	p=4	p=5	p=6	p=7
CPU core time per element [s]	0.006	0.103	1.164	6.543	33.834	125.039	390.167
GPU multiprocessor time per element [s]	0.006	0.046	0.466	6.596	46.271	257.341	889.719
Speed-up (1 core, 1 multiprocessor)	1.07	2.25	2.50	1.00	0.73	0.49	0.44
Speed-up (2 cores, 16 multiprocessors)	8.56	18.00	19.98	7.97	5.85	3.89	3.51

Table 5: Execution times in milliseconds and speed-up GPU versus CPU for the numerical integration algorithm applied to the model problem with different orders of approximation p for GeForce 8800GTX GPU and AMD X2 CPU

Table 5 shows the results of experiments. First, execution times are presented for a single CPU core and a single GPU multiprocessor. The numbers are obtained by dividing the execution times for the whole CPU and GPU by the number of cores or multiprocessors. The procedure is justified by the fact that the numerical integration calculations are fully scalable - for large enough numbers of finite elements (the only interesting case) execution time for p cores or multiprocessors will be p-times smaller than for one core or multiprocessor. Hence for any real CPU and GPU architectures, with the same cores or multiprocessors, execution times can be computed from results in table 5. Moreover, the results can be used to compute GPU versus CPU speed-ups. The speed-up results in table 5 for the reference processors: NVIDIA GeForce 8800GTX graphics card and AMD X2 CPU were obtained by dividing CPU times by two and GPU times by 16 and then computing speed-up in the standard way. The same procedure can be repeated for different CPUs and GPUs, since in current designs the performance of single cores does not change that much, while the number of utilized cores changes in time and from one model to another.

6. Conclusions

The most important conclusion is that still the most limiting factor for GPU computations of the kind presented in the paper are the small resources available to individual threads. Given possible massive parallelism in the algorithm considered in the paper, it was not fully realized in practice. Because of insufficient memory resources we could not choose the optimal strategy for parallelization and had to, instead, perform additional splittings of data structures as well as additional calculations.

We succeeded however in another difficult task, designing data structures in such a way so to allow for proper (coalesced) accesses of threads to arrays. As a final result, the performance obtained during computational experiments is encouraging and it seems that finite element codes, also those utilizing higher order of approximation, should profit from porting to GPU accelerators another important algorithm – numerical integration.

Acknowledgment

The support of this work by the Polish Ministry of Science and Higher Education under the grant N N501 120836 is gratefully acknowledged

References

[1] D. Göddeke, H. Wobker, R. Strzodka, J. Mohd-Yusof, P. McCormick, S. Turek, Co-processor acceleration of an unmodified parallel solid mechanics code with FEASTGPU, International Journal of Computational Science and Engineering (IJCSE) (2009) to appear.
[2] P. Ciarlet, The Finite Element Method for Elliptic Problems, North-Holland, Amsterdam, 1978.
[3] NVIDIA CUDA Programming Guide 2.2.1, NVIDIA, 2009.
[4] M. Baboulin, A. Buttari, J. Dongarra, J. Kurzak, J. Langou, J. Langou, P. Luszczek, S. Tomov, Accelerating scientific computations with mixed precision algorithms, *preprint*.
[5] D. Göddeke, R. Strzodka, S. Turek, Performance and accuracy of hardware-oriented native-, emulated- and mixed-precision solvers in FEM simulations, International Journal of Parallel, Emergent and Distributed Systems 22 (4) (2007) 221–256.
[6] D. Komatitsch, D. Micha, G. Erlebacher, Porting a high-order finite-element earthquake modeling application to NVIDIA graphics cards using CUDA, Journal of Parallel and Distributed Computing 69 (5) (2009) 451–460.