Proceedings of the International Multiconference on Computer Science and Information Technology pp. 337–342
ISBN 978-83-60810-27-9
ISSN 1896-7094

Higher order FEM numerical integration on GPUs with OpenCL

[1]Przemysław Płaszewski, [12]Krzysztof Banaś, [2]Paweł Macioł
[1]AGH University of Science and Technology,
Department of Applied Computer Science and Modelling,
al. Mickiewicza 30, 30-059 Kraków, Poland
[2]Cracow University of Technology,
Insitute of Computer Modelling,
ul. Warszawska 24, 31-155 Kraków, Poland

***Abstract*—Paper presents results obtained when porting FEM 2D linear elastostatic local stiffness matrix calculations to Tesla architecture with OpenCL framework. Comparison with native NVIDIA CUDA implementations has been provided.**

I. Motivation

IN FINITE element simulations usually two computation stages most significantly impact performance of the whole process:

- obtaining global stiffness matrix
- solving system of linear equations

In case of non-linear, higher order element geometry, higher order approximations and p and hp-adaptations, process of obtaining global stiffness matrix requires computationally intensive separate calculations of local (element) stiffness matrices. For some problems it can be most time consuming stage of FEM calculations.

Aim of our work is parallelizing this stage utilizing modern graphics processor units (GPUs) and OpenCL platform.

The paper is organized as follows. The first two sections are devoted to the definition of the problem of FEM numerical integration. In the third section we summarize the problem in terms of requirements for numerical algorithms. Then we show how to design parallel algorithms that solve the formulated computational problem on modern GPUs. The results of experiments close the paper.

II. Finite element numerical integration

The standard procedure in FEM computations consist in obtaining weak formulation of a problem, discretizing problem domain Ω into finite elements and utilizing appropriate basis functions—constructed from element shape functions—to create a system of linear equations, with the global stiffness matrix as the system matrix, that is then solved to provide approximate solution.

Generation of global stiffness matrix is usually performed by calculating integrals over finite elements, then assembling obtained that way local matrices into global one. Since integrals evaluation over multiple different element geometries (possibly curved) would pose a problem, elements are mapped to a reference element with simple geometry and integrals are calculated over its area. Every element is processed independent of the others thus many can be calculated in parallel.

III. Model problem

We chose 2D linear elastostatics problem. As a model finite element we use quadrilateral with curved, second order, geometry. Solution is approximated with hierarchical shape functions up to order $p = 7$, constructed from tensor products of 1D Lobato hierarchical functions [2]. Reduced space was used with number of shape functions equal to

$$n = 4 + 4(p-1)_+ + (p-2)(p-3)_+/2$$

where q_+ denotes $max(q, 0)$. Local stiffness matrix dimension is equal to $2n$, where n is number of shape functions for a particular order p. Matrix entries are results of calculating the integral [1]

$$k_{IJ}^{(e)} = \iint_{\Omega_{ref}} (([D^*]\{\varphi_I\})^T[E][D^*]\{\varphi_J\}) \mid J \mid d\xi d\eta \quad (1)$$
$$I, J = 1, 2, ..., 2n$$

where $k_{IJ}^{(e)}$ is (I, J) matrix entry, Ω_{ref} is $[-1, 1] \times [-1, 1]$ reference quadrilateral, $[E]$ is 3×3 material matrix, $\mid J \mid$ is the determinant of the Jacobian matrix of geometry transformation, $\{\varphi_I\}$ and $\{\varphi_J\}$ are columns of $2n \times 2$ matrix of shape functions. $[D^*]$ is the matrix differential operator

$$[D^*] = \begin{bmatrix} \frac{\partial}{\partial x} & 0 \\ 0 & \frac{\partial}{\partial y} \\ \frac{\partial}{\partial y} & \frac{\partial}{\partial x} \end{bmatrix}$$

where

$$\begin{aligned} \frac{\partial}{\partial x} &= J_{11}^* \frac{\partial}{\partial \xi} + J_{12}^* \frac{\partial}{\partial \eta} \\ \frac{\partial}{\partial y} &= J_{21}^* \frac{\partial}{\partial \xi} + J_{22}^* \frac{\partial}{\partial \eta} \end{aligned}$$

and $[J^*]$ is inverse Jacobian matrix.

Integration over reference quadrilateral is done using Gauss quadratures where the double integral is replaced by double sum

$$\int_{-1}^{1}\int_{-1}^{1} f(\xi, \eta) d\xi d\eta \approx \sum_{i=1}^{n_g}\sum_{j=1}^{n_g} f(\xi_i, \eta_j) W_i W_j$$

In case of bilinear mapping, integration is accurate when number of gauss points n_g in both dimensions is not less than $(p+1)^2$ where p is order of approximation (i.e. highest polynomial order in shape functions set).

IV. Computation overview

Evaluating the double integral boils down to calculating double sum, which can be achieved with single loop over gauss points in two dimensions. In every iteration, for every entry in a matrix, the result of calculating (1) is multiplied by Gauss weights and added to the (first zeroed) element stiffness matrix matrix.

The integrated formula differs for different matrix entries only in shape functions used—for every entry a distinct pair of shape functions is evaluated (taking into account the symmetry of a matrix).

Jacobian determinant and the inverse of Jacobian matrix (which appears in $[D^*]$ operator) are the same for all matrix entries thus can be calculated once per iteration.

Assuming constant Young modulus and Poisson ratio over a single finite element, material matrix $[E]$ can be calculated once per element.

Calculation of single local stiffness matrix requires:

- single calculation of material matrix $[E]$
- n_g calculations of Jacobian determinant and inverse of the Jacobian matrix, where n_g is number of gauss points
- $n_g \frac{n(n+1)}{2}$ calculations of entries (1), where n is matrix dimension (symmetric matrix case)

V. Parallel numerical integration in OpenCL

Finite element local stiffness matrices are calculated independent of each other therefore can be carried out concurrently. The process of calculating a single local stiffness matrix can also be parallelized by simultaneous computations of its entries. Therefore we can identify two possible parallelization levels:

- processing finite element matrices
- processing matrix entries

This naturally fits to OpenCL parallelization model [4] of:

- work-groups executed concurrently, independent of each other
- work-items executed concurrently, cooperating inside each work-group

Our implementation takes advantage of above similarities and divides computations as follows:

- each work-group is responsible for processing single finite element
- work-item is responsible for calculating one or more matrix entries

Despite the fact that the OpenCL platform aims to provide unified execution environment for broad range of hardware solutions like multi-core CPUs, graphics processors and accelerators like IBM Cell, programmer still needs to realize differences between them and optimize his code accordingly. Our implementation targets Nvidia Tesla architecture.

Calculations on device, in our case GPU, are directed by host CPU. Its role is to upload elements data into device global memory, execute kernel and download output matrices when kernel finishes. If not all elements can be processed in single kernel launch—for example due to limited amount of global device memory—whole computations can be divided across multiple sequences of:

- uploading data to device memory
- executing kernel
- downloading data

Kernel launch is non-blocking therefore host can process remaining batches of elements while waiting for GPU to finish, thus increasing overall performance.

Parallel calculations overview is presented on Fig.1.

Each work-group reads the data representing particular finite element from global device memory. Then work-items in cooperation evaluate matrix entries and store them in local memory—shared by all items in a work-group. When done, matrix is copied from local to global device memory accessible by host.

Data that are common for all finite elements i.e. gauss points and weights is calculated once on CPU, stored in constant memory and available for all work-groups. Constant memory is cached on a GPU and well serves the purpose of supplying the multiprocessors with the data that are the same for all elements of a given order p.

During the execution work-groups are queued and successively consumed by device computation units. Each computation unit—in our case a GPU multiprocessor—can execute more than one work-group at a time, however splitting single work-group across many computation units is not allowed.

The number of simultaneously processed work-groups on single computation unit (resident work-groups) depends on factors like consumption of registers, consumption of multiprocessor local memory and size of a work-group. GPU scheduling hardware will send as much work-groups as possible for concurrent execution on single multiprocessor until there are no resources left. In situation when number of consumed registers, shared memory or number of work-items exceeds the half of maximum available for particular hardware, only one work-group at a time will be resident on single multiprocessor.

Optimal use of multiprocessor resources is important to achieve high occupancy i.e. high number of work-items from one or more work-groups executed concurrently on multiprocessor. In our case resource consumption depends on approximation order—the higher is the order p the bigger is the local stiffness matrix and more local memory is needed by work-group during the computations.

Resource limits for Nvidia GPU devices of compute capability 1.3 (all cards we tested) are:

- Multiprocessor registers: 16384
- Local memory per multiprocessor: 16KiB
- Maximum number of work-items in work-group: 512
- Maximum number of resident work-items on multiprocessor: 1024

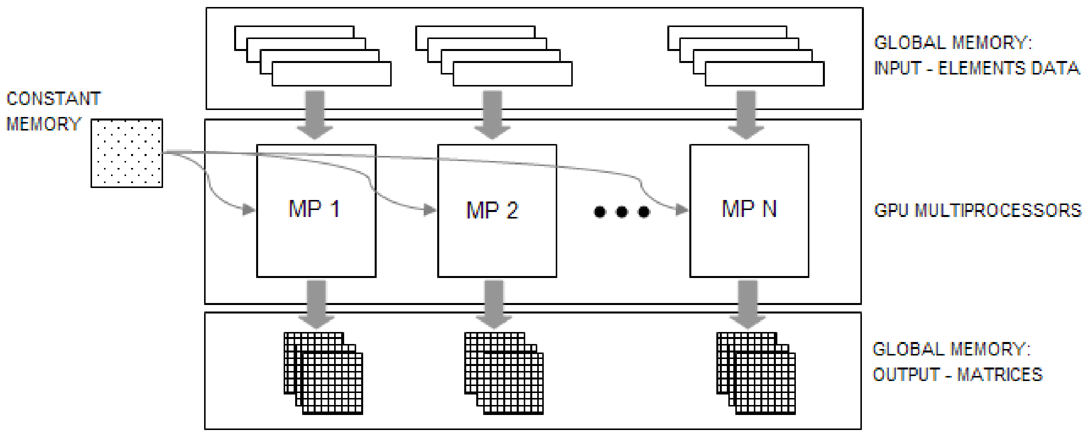

Fig. 1. Parallel calculations on GPU.

Available registers are uniformly distributed among resident work-items executed on multiprocessor and local memory is divided across resident work-groups.

In order to generate local stiffness matrix for single finite element, following information needs to be supplied (23 float values in total):

- geometry of quadrilateral (coordinates of its vertices and second order geometry mapping nodes)
- edge orientations
- material information (Young modulus, Poisson ratio)

We have padded each element data to 32 and placed all elements data in contiguous region of global memory in order to achieve specific coalesced access pattern [3] by every work-group. For the same reason we also used padding for output matrices.

Every work-group (i.e. all its work-items executing in parallel) performs single coalesced read from input global memory region in the beginning of computations to get element data and coalesced writes at the end of execution to store generated matrix.

Total size of input data is

$$32 * sizeof(float) * numelems$$

Total size of output matrices is

$$matpaddedsize(p) * sizeof(float) * numelems$$

where $matpaddedsize(p)$ is matrix size with padding and depends on order of approximation p. Due to symmetry we are only calculating and storing upper-half matrices, thus reducing both local and global memory usage. Sizes of matrices for different approximation orders are presented in table I.

Depending on the order p different kernel is being executed on the device. Flow of computation for every kernel is similar. The differences come from different number of gauss points, different set of hierarchical shape functions and different sizes of matrices and manifest themselves in:

- amount of local memory needed to compute stiffness matrix
- number of iterations over gauss points
- amount of local memory needed to compute derivatives of shape functions in every Gauss point
- number of matrix entries per work-item
- number of work-item engaged in calculations of stiffness matrix (which is less-or-equal to the work-group size)

For every kernel we configured computations in such way not to exceed the available computation unit resources (registers and local memory), maximize global memory throughput with coalesced reads and writes, maximize computation unit occupancy and minimize divergent branches inside groups of 32 consecutive work-items called warps.

Execution configuration included proper sizing of work-groups and defining number of matrix entries calculated by single work-item. We experimented with broad range of configurations for every kernel in order to chose optimal ones—those are presented in table I.

In case of higher p orders, limited multiprocessor resources—especially local memory size of only 16KiB on Tesla architecture—prevented simultaneous execution of more than one work-group thus reducing multiprocessor occupancy.

We implemented kernels for orders of approximation up to 7—for higher orders half-matrix size is bigger than amount of local memory available on Tesla architecture and different approach to computations would be needed.

VI. Kernel computations

Main steps of kernel execution are:

1) Zero matrix in local memory (all work-items)
2) Load element data from global memory (first warp)
3) Calculate 3×3 material matrix $[E]$ (single work-item)
4) Loop over gauss points:
    a) Calculate values of shape functions derivatives (single work-item)

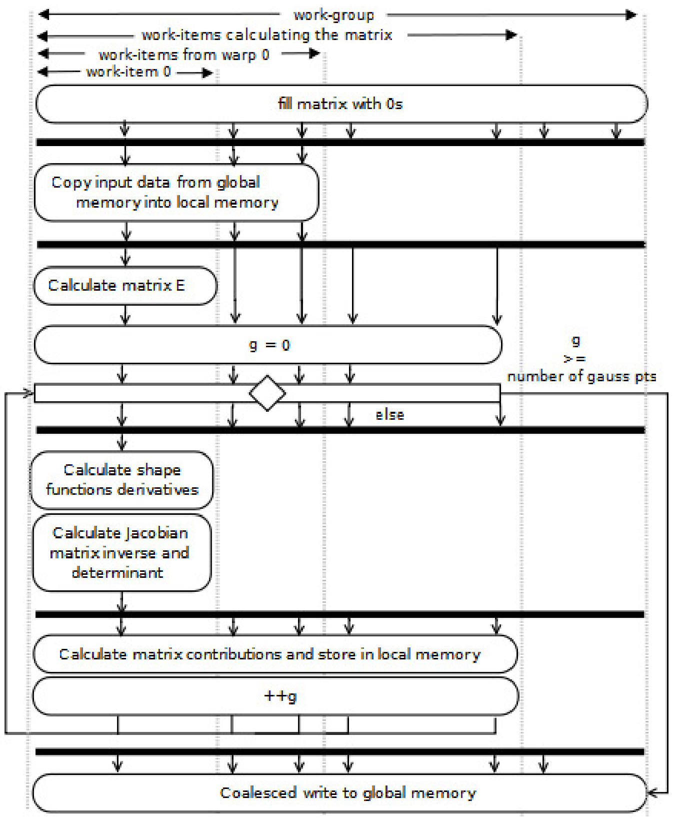
work-group
work-items calculating the matrix
work-items from warp 0
work-item 0
fill matrix with 0s
Copy input data from global memory into local memory
Calculate matrix E
g = 0
g >= number of gauss pts
else
Calculate shape functions derivatives
Calculate Jacobian matrix inverse and determinant
Calculate matrix contributions and store in local memory
++g
Coalesced write to global memory

Fig. 2. Work-group execution.

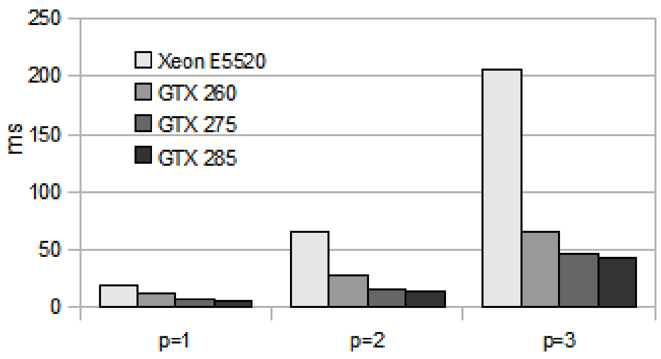

Fig. 4. Execution times for 10000 elements.

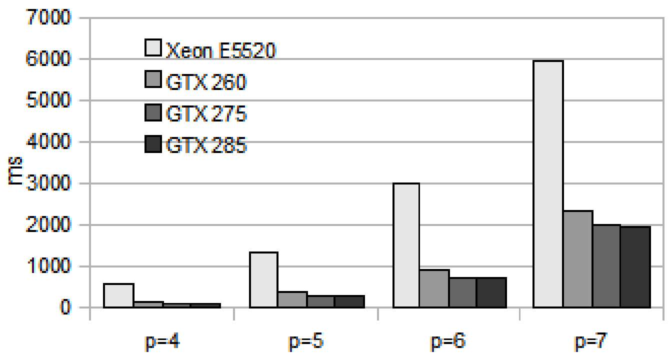

Fig. 5. Execution times for 10000 elements (continued).

b) Calculate Jacobian determinant and the inverse of Jacobian matrix (single work-item)
c) Calculate stiffness matrix contributions (in parallel—number of work items depends on kernel—see table I)

5) Upload matrix to global memory (all work-items)

Fig. 2 presents the calculation flow for a work-group. Black horizontal bars indicate OpenCL $CLK\_LOCAL\_MEM\_FENCE$ synchronization barriers. Those are needed to ensure that all calculations from previous steps are done before utilizing their results in steps that follows. We experimented with disabling synchronizations in order to observe their influence on overall algorithm performance—execution times dropped by not more than 5%.

Material matrix $[E]$, Jacobian determinant, inverse of Jacobian matrix and shape function calculations never took more than 10% of total execution time. Attempts to spread shape functions calculations across more work-items and perform them in parallel resulted in higher register consumption per work-item thus reducing occupancy and decreasing performance for most kernels.

In order to effectively utilize limited local memory, local stiffness half-matrix is stored in linear contiguous fashion row by row, every consecutive row being one shorter than previous. Every work-item in a work-group is identified by its local id obtained with get_local_id device function. Assignment of matrix entries to distinct work-items relies on mapping formula involving square root operation. Fig. 3 presents work-items assignments for kernel $p = 3$. For example work-item with id 44 is responsible for calculating five matrix entries at positions (22,11), (23,11), (12,12), (13,12) and (14,12). Work-items with numbers from 60 to 63 are left idle during matrix calculations, but do participate in writing padded matrix to global memory when done.

Matrices in every kernel are padded to the nearest multiply of 32 and their size is multiply of work-group size to achieve coalesced writes.

VII. Results

Figures 4 and 5 show results we obtained while calculating 10^4 matrices on 3 NVDIA GTX series graphics cards. For comparison we performed tests of cache optimized sequential code run on single Nehalem core of Intel Xeon E5520 processor.

Best performing kernel ($p = 4$) achieved 5.3 speedup over sequential implementation. Lowest speedups of 3.3 and 3 were observed with kernels for $p = 1$ and $p = 7$ respectively.

For GPU implementation timings include transfers of data between host and device through PCI-Express bus ($clEnqueuReadBuffer$ and $clEnqueueWriteBuffer$ host functions). Data transfer is amortized—especially for lower order kernels—only when number of processed elements is high enough—for example for $p = 1$ and 1000 elements transfers take 25% of total execution time while for $p = 5$ only 4%.

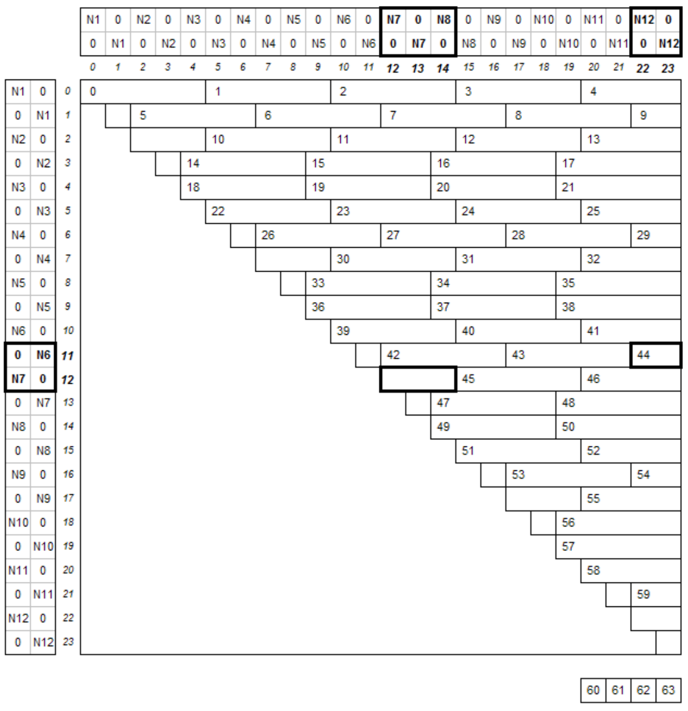

Fig. 3. Work-items to matrix entries assignment for $p = 3$.

TABLE I
KERNEL PARAMETERS

	p=1	p=2	p=3	p=4	p=5	p=6	p=7
Work-group size	64	192	64	128	192	384	448
Work-items involved in matrix generation	36	136	60	119	181	366	418
Matrix dimension	8	16	24	34	46	60	76
Padded matrix entries number	64	192	320	640	1152	1920	3136
Register consumption per work-item	12	13	18	19	20	19	18
Local memory consumption per work-group (bytes)	568	1208	1848	3288	5528	8824	13944

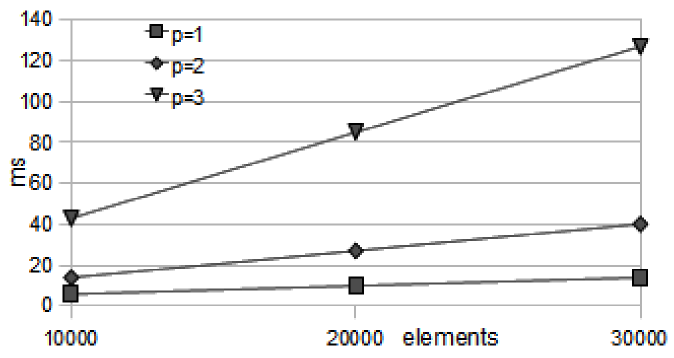

Fig. 6. Execution times for different elements number on GTX 285.

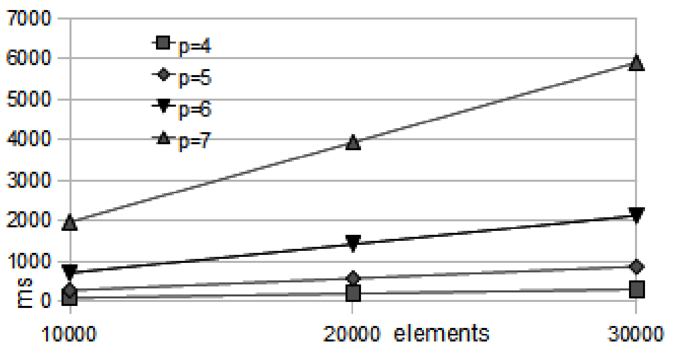

Fig. 7. Execution times for different elements number on GTX 285 (continued).

Figures 6 and 7 present execution times with varying number of elements processed. As expected execution time scales in linear fashion with elements number—as every element is represented on device by work-group and work-groups are queued for execution on multiprocessors.

Of 3 cards tested 2 (GTX 275 and GTX 285) have the same number of multiprocessor (32) and differ in global memory bus bandwidth—448 bit and 512 bit respectively. Performance difference between those two cards is minimal as compared to

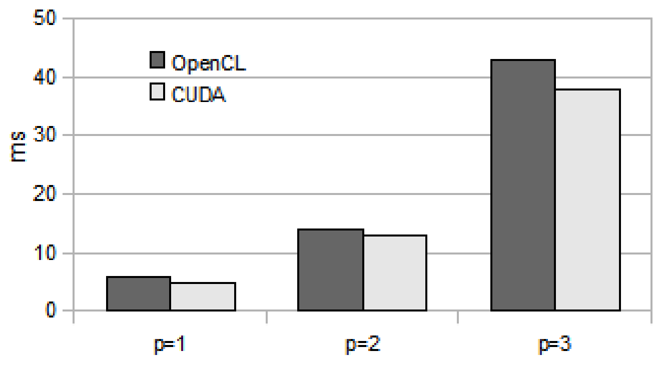

Fig. 8. Performance comparision of OpenCL and CUDA with 10000 elements on GTX 285.

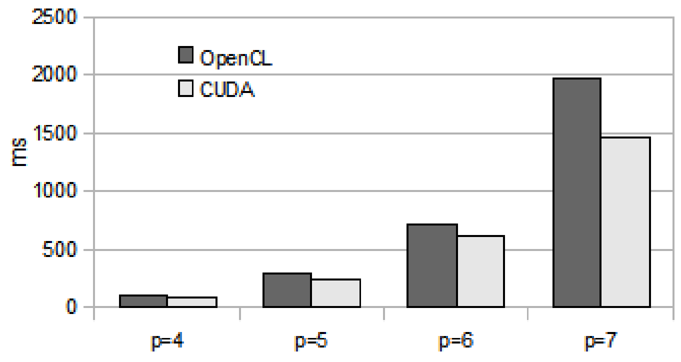

Fig. 9. Performance comparision of OpenCL and CUDA with 10000 elements on GTX 285 (continued).

the difference between cards with different number of multiprocessors (GTX 260 with 27 and GTX 275 and GTX 285 both having 30). Our implementation isn't memory constrained since all operations except single input and output coalesced writes are performed on local multiprocessor memory and the biggest size of transferred batch is only 13KiB for matrix of $p = 7$.

One can expect linear performance scaling with increasing number of GPU multiprocessors. Number of simultaneously processed work-groups on all multiprocessors is given as:

$$R_{wg} \times M$$

where R_{wg} is number of resident work-groups per multiprocessor and M number of multiprocessors. Let t_1 be the time that takes to process $R_{wg} \times M_1$ work-groups on GPU G_1 having M_1 multiprocessors. Similarly t_2, M_2 for GPU G_2. Assuming identical performance (i.e same clock rate and architecture) of multiprocessors in both GPUs $t_1 = t_2 = t$. Total time of processing N work-groups on GPU G_1 is:

$$T(G_1, N) = \frac{N}{M_1 \times R_{wg}} t$$

and on G_2:

$$T(G_2, N) = \frac{N}{M_2 \times R_{wg}} t$$

Speedup calculated as a ratio of those two times:

$$\frac{T(G_1, N)}{T(G_2, N)} = \frac{\frac{N}{M_1 \times R_{wg}} t}{\frac{N}{M_2 \times R_{wg}} t} = \frac{M_2}{M_1}$$

indicates linear scaling with number of multiprocessors.

According to the above statement observed speedup should be not less than 11% between GTX 260 and GTX 275 processors. Our benchmarks demonstrate speedup in fact being higher—especially for lower orders of approximation—due to increased GPU clock-rate and better system components (i.e. CPU, motherboard) on machine equipped with GTX 275 card.

We compared OpenCL performance with native NVIDIA CUDA 3.0 implementations (see Figures 8 and 9) of our kernels. For $p = 1$ and $p = 2$ kernels were compiled with same register consumptions on both platforms and results are close. Initial compilation of other kernels resulted in increased register usage of OpenCL kernels as compared to CUDA.

Since higher register consumption resulted in much lower occupancy and significantly decreased performance we forced OpenCL compiler to use the same number of registers as in the CUDA build. Consumption decreased, but unfortunately OpenCL compiler was unable to achieve this without spills to slow private memory thus slightly decreasing performance (however not that much as with increased register consumption).

Overall performance of OpenCL in our case is comparable to CUDA, except when compiler is not able to optimize its output the way more mature CUDA tools do.

REFERENCES

[1] Barna Szabo and Ivo Babuska, *Finite Element Analysis,* Wiley-Interscience; 1991.
[2] Pavel Solin and Karel Segeth and Ivo Dolezel, *Higher-Order Finite Element Methods,* Chapman & Hall/CRC; 2003.
[3] Mark Harris, "Optimizing CUDA," *in Supercomputing conference,* Reno, NV, 2007.
[4] Khronos OpenCL Working Group, *The OpenCL Specification 1.0,* 2009.