Efficient Vectorised Cuda Kernels for High-Order Finite Element Flow Solvers

Jan Eichstädt[1,*], David Moxey[2], Joaquim Peiró[1]

[1] Department of Aeronautics, Imperial College London, UK. jan.eichstaedt13@imperial.ac.uk, j.peiro@imperial.ac.uk

[2] College of Engineering, Mathematics and Physical Sciences, University of Exeter, UK. d.moxey@exeter.ac.uk

Abstract

In this work, we develop efficient kernels for elemental operators of matrix-free solvers of the Helmholtz equation, which are the core operations for more complete Navier-Stokes solvers. We consider straight-sided and deformed quadrilateral elements from unstructured high-order meshes. We investigate two types of efficient *CUDA* kernels for a range of polynomial orders; a first type which maps each elemental operation to a *CUDA*-thread, and a second that maps each element to a *CUDA*-block. Our results show that the first option is beneficial for small elements with low polynomial order, whereas the second option is beneficial for larger elements. For both options we show the importance of the right layout of data structures, and analyse the effect of utilising different memory spaces on the GPU.

Key words: *High-order FEM; Parallelisation; CUDA*

1 Introduction

Modern shared-memory systems like GPUs offer massively parallel compute power, while the gap between available FLOPS and memory bandwidth increases. This demands for parallel schemes with high arithmetic intensity, such as high-order finite element methods. These methods are frequently used to simulate fluid dynamics problems, for example by discretising the Navier-Stokes equation and using the large eddy simulation (LES) approach to model the turbulent nature of the flow. High-order methods are very suitable for this class of simulations, due to their low diffusion and dispersion error and their exponential error convergence [3]. Considering the incompressible Navier-Stokes equation, an operator splitting scheme as introduced in Reference [2] leads to very efficient implementations. In this scheme, the advection terms are solved using explicit time-stepping, while the diffusion and pressure terms are solved using implicit time-stepping. The latter forms the computationally most expensive part and can be expressed in terms of solving the Helmholtz equation. In a continuous Galerkin projection, the global Helmholtz operator can be efficiently solved with a low memory footprint using a *matrix-free* approach, which is performed in three steps: first the global modes are scattered to their local element-wise modes, then the Helmholtz operation is performed on each element independently, and finally all elemental modes are gathered back to their global modes. In this work we focus on the central part of this scheme, the parallel execution of the elemental Helmholtz operator. We restrict our work to quadrilateral elements, both straight-sided and deformed, but note that our investigations can be easily transferred to other tensor product based element types. Similar work has been conducted by other finite element groups, see for example *deal.II* [4] and *Dune* [1] focusing mostly on efficient vectorised operations for quadrilateral and hexahedral elements on CPUs.

Deformed high-order elements allow to represent curved CAD geometries exactly, leading to more accurate flow solutions. They however require more involved computations, as the Jacobian of these elements is not constant, a fact that on the other hand we exploit for straight-sided elements. The evaluation of the elemental Helmholtz operator in described in more detail in Section 2. It is crucial to design our algorithms with the GPU hardware in mind, especially with regards to the hierarchical parallel structure and also with regards to the different memory spaces, that can be addressed explicitly. We discuss this in more detail in Section 3. High-order finite element methods additionally posses the desirable feature of varying the polynomial order p of the shape functions that are used to perform the elemental evaluations. The question which polynomial order to choose is influenced by multiple factors and is subject of significant ongoing research efforts. Here we consider the computational aspect, thus our target measure is maximising a throughput expressed

in degrees of freedom (DOF) per time-unit. With increasing polynomial order the number of operations per DOF increases faster than the memory footprint. This results in higher arithmetic intensities of the algorithms for higher polynomial orders p, which on GPUs is often required to achieve good performance and utilisation of the hardware. We present performance results of our kernels for a range of polynomial order in Section 4 and draw conclusions in Section 5.

2 Method

We implement the following arithmetic operations in our *CUDA* kernels. The elemental Helmholtz operation $\boldsymbol{H}^e$ for a quadrilateral element is the sum of the Laplacian operator and the mass operator and can be expressed in matrix form as

$$\hat{s}_l^e = \boldsymbol{H}^e \hat{u}_l^e = [\boldsymbol{L}^e + \lambda \boldsymbol{M}^e] \hat{u}_l^e \tag{1}$$

$$\hat{s}_l^e = [(\boldsymbol{D}_{x_1}\boldsymbol{B})^T \boldsymbol{W} \boldsymbol{D}_{x_1}\boldsymbol{B} + (\boldsymbol{D}_{x_2}\boldsymbol{B})^T \boldsymbol{W} \boldsymbol{D}_{x_2}\boldsymbol{B} + \lambda \boldsymbol{B}^T \boldsymbol{W} \boldsymbol{B}] \ \hat{u}_l^e \tag{2}$$

Here, the basis matrix $\boldsymbol{B}$ with $B_{i,j} = \phi_j(\xi_i)$ denotes the backward-transform from coefficient space to physical space, its transform $\boldsymbol{B}^T$ is the forward operator from physical space to coefficient space. The diagonal weight matrix $\boldsymbol{W}$ with $W_{i,j} = J_{i,j} w_i w_j$ includes the Jacobian J, that is the mapping between the standard element defined on $\xi \in [-1, 1]$ and the elemental global coordinates x, as well as the quadrature weightings w_i and w_j. The derivative matrices $\boldsymbol{D}_{x_1}$ and $\boldsymbol{D}_{x_2}$ with respect to the two spatial directions x_1 and x_2 are

$$\boldsymbol{D}_{x_1} = \boldsymbol{\Lambda} \left(\frac{\partial \xi_1}{\partial x_1} \right) \boldsymbol{D}_{\xi_1} + \boldsymbol{\Lambda} \left(\frac{\partial \xi_2}{\partial x_1} \right) \boldsymbol{D}_{\xi_2} \tag{3}$$

$$\boldsymbol{D}_{x_2} = \boldsymbol{\Lambda} \left(\frac{\partial \xi_1}{\partial x_2} \right) \boldsymbol{D}_{\xi_1} + \boldsymbol{\Lambda} \left(\frac{\partial \xi_2}{\partial x_2} \right) \boldsymbol{D}_{\xi_2}, \tag{4}$$

with the diagonal coefficient matrices $\boldsymbol{\Lambda}$, and the block-diagonal derivative matrices $\boldsymbol{D}_{\xi_1} = \frac{\partial}{\partial \xi_1}$ and $\boldsymbol{D}_{\xi_2} = \frac{\partial}{\partial \xi_2}$. We precompute the two matrix products $\boldsymbol{D}_{\xi_1}\boldsymbol{B}$ and $\boldsymbol{D}_{\xi_2}\boldsymbol{B}$, that are the same for all elements. To minimise the operator count for an efficient implementation, the Jacobian matrix and the four different coefficient matrices $\boldsymbol{\Lambda}$ for each element have also been precomputed and stored. We note that for straight-sided elements these five matrices are constant for each element, which we leverage in our kernel implementations. We employ tensor-product bases for our quadrilateral elements, which further allows the exploitation of the sum-factorisation approach that reduces the number of operations.

3 Implementation

Within the global Helmholtz operation, each elemental operation can be performed independently and hence in parallel. Additionally, we can realise a second nested level of parallelism within each element, over the different quadrature points or modes. We hence investigate two different mappings of the parallel algorithms to the parallel GPU hardware. As the first option, each elemental operation is mapped to a single *CUDA*-thread. Here it is paramount to place special emphasis on efficient data-structures by interleaving the data of a group of 32 elements for the vectorised operations within a *CUDA*-warp. This is illustrated in Figure 1 for the default memory layout, where each elemental data set is placed consecutively into the linear memory array. Here all the first data units of each elemental data set are scattered over the memory, and need to be loaded individually. However, when the data is interleaved with a stride equal to the vectorisation length, as illustrated in Figure 2, a contiguous memory access can be achieved, and the bandwidth of loading operations is increased significantly. As the second option, each elemental operation is mapped to a single *CUDA*-block. Here it is critical to harness the parallelism within the elemental operations by assigning each quadrature point or mode (depending if the operation is in physical or modal space) to one *CUDA*-thread. In this case the default memory layout is employed.

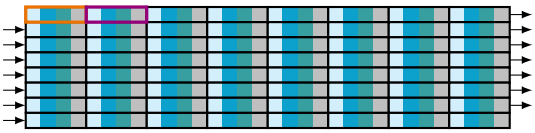

Figure 1: Default memory layout with stride = 1

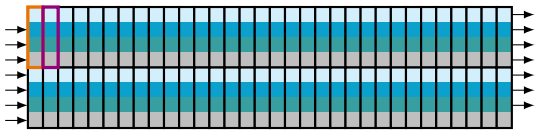

Figure 2: Interleaved memory layout with stride = 32 (size of GPU warp)

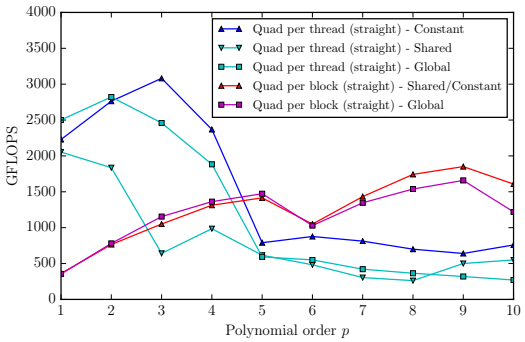

Figure 3: Performance of various implementations for straigh-sided quadrilateral elements

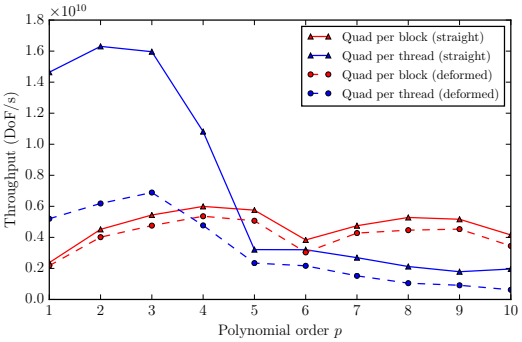

Figure 4: Performance of *optimal* implementations for straigh-sided and deformed quadrilateral elements

To address a second design consideration, we analyse the effect of placing utility data like the basis matrix $\boldsymbol{B}$, quadrature weights w, and derivative matrices $\boldsymbol{D}_{\xi_i}$ into either *global* memory, *constant* memory, or *shared* memory of the GPU. *Global* memory has the lowest memory bandwidth, whereas *shared* memory is physically located on a higher cache level with increased bandwidth, but smaller size. Here each thread can fetch data without a penalty for non-contiguous access. Finally, the *constant* memory allows very fast read access only, if all threads within one warp are accessing the same memory address. This makes this memory space a promising candidate for utility data of the completely vectorised kernels.

4 Results

We investigate our templated Helmholtz kernels for polynomial orders of $p = 1$ to $p = 10$ on a Nvidia Titan V GPU, which possesses a maximum of 6900GFLOPS. We utilise the *nvcc* compiler version 9.1 and set the *-O3* optimisation flag. In Figure 3 we report the average achieved FLOPS over 1000 executions of five different kernels for straight-sided quadrilateral elements. It shows that placing constant utility data into the *constant* memory space for the completely vectorised operation can lead to performance improvements of up to a factor of two, compared with the default of loading this data from *global* memory space. For elements of $p > 6$, that are mapped to a *CUDA*-block performance improvements of up to 20% can be realised by loading such utility data to the *shared* memory space. In Figure 4 we compare the optimum kernels in terms of memory placement. We can derive that for the specific GPU employed here, the mapping of elemental operations per thread is beneficial up to a polynomial order of $p = 4$, after which the registers are overflowing into the slower *local* memory. For higher polynomial orders the mapping per block is superior, whereas for low polynomial orders there are not enough modes to fully occupy all 32 threads of a warp. The figure further shows the different throughput achieved depending on the deformation of the elements.

To assess how efficiently we employ the GPU hardware, we plotted the achieved FLOPS of the four optimal kernels in two different so called *roofline* plots. (The horizontal lines depict the limits in terms of FLOPS, whereas the diagonal line depicts the memory bandwidth limit.) From Figure 5, we can deduce that the kernels that map per thread are memory bound at the DRAM

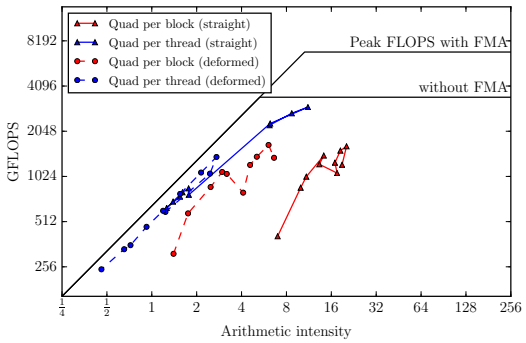

Figure 5: Rooflines based on DRAM memory

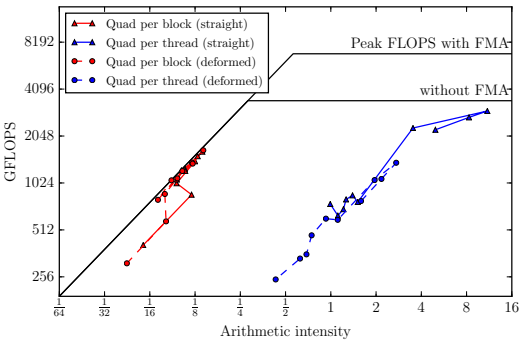

Figure 6: Rooflines based on L1 cache

level, whereas from Figure 6 we can deduce that the kernels that map per block are memory bound at the L1 cache. The best kernels nevertheless achieve 20% to 42% of the maximum theoretical FLOPS, indicating a very good utilisation of the GPU hardware.

5 Conclusions

We have compared two options of mapping the parallel execution of the Helmholtz operator to the parallel GPU hardware using the *CUDA* programming model. The first option maps each elemental operation to one *CUDA*-thread, whereas the second options maps each elemental operation to a *CUDA*-block, where we assign each quadrature point or mode to a *CUDA*-thread. Our results show that the first option is beneficial for small quadrilateral elements with low polynomial order of up to $p = 4$, as larger elements quickly approach the register limits of the GPU. The second option is beneficial for larger elements, that provide high arithmetic intensity to fully occupy all *CUDA*-threads.
Our investigation shows that placing constant utility data into the *constant* and *shared* memory space leads to performance gains compared with only using *global* memory space.
Finally, straight-sided elements result in a substantially increased throughput compared with deformed elements, which should be taken into account in mesh-optimisation algorithms.

Acknowledgments

J.E. gratefully acknowledges the support of his research through the President's Scholarship of Imperial College London.

References

[1] P. Bastian, C. Engwer, J. Fahlke, Ma. Geveler, D. Göddeke, O. Iliev, O. Ippisch, R. Milk, J. Mohring, S. Müthing, M. Ohlberger, D. Ribbrock, and S. Turek. Hardware-Based Efficiency Advances in the EXA-DUNE Project. In Hans-Joachim Bungartz, Philipp Neumann, and Wolfgang E Nagel, editors, *Software for Exascale Computing - SPPEXA 2013-2015*, pages 3–23, Cham, 2016. Springer International Publishing.

[2] G. E. Karniadakis, M. Israeli, and S. A. Orszag. High-Order Splitting Methods for the Incompressible Navier-Stokes Equations. *Journal of Computational Physics*, 97(2):414–443, 1991.

[3] G.E. Karniadakis and S.J. Sherwin. *Spectral/hp Element Methods for Computational Fluid Dynamics*. Oxford University Press, 2nd ed. edition, 2005.

[4] M. Kronbichler and K. Kormann. Fast Matrix-Free Evaluation of Discontinuous Galerkin Finite Element Operators. *ACM Transactions on Mathematical Software*, 45(3):1–40, 2019.