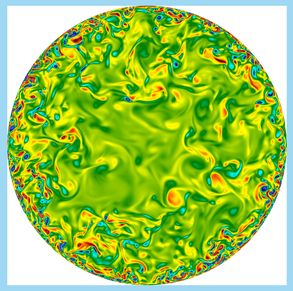
Optimization of Tensor-product operations in Nekbone on GPUs

Martin Karp, Niclas Jansson, Artur Podobas, Philipp Schlatter and Stefano Markidis

KTH Royal Institute of Technology and Swedish e-Science Research Centre (SeRC)

Overview

- The optimization on GPUs of the most important tensor-product, the Poisson operator, in Nekbone is studied. This operation dominates the compute time in the larger CFD solver Nek5000 as well as Nekbone and performance improvements in this part of the solver have a large impact on the runtime. Optimizations performed in Nekbone can then be transferred to the entire solver.
- Interfacing with the legacy Fortran codebase is done with a mixed OpenACC and CUDA approach where the kernel for the Poisson evaluation is implemented in CUDA.
- The optimization is done with a different 2D-thread structure proposed by Świrydowic et. al [3] that enables loop unrolling as well as increased register utilization.



Simulation of turbulent flow in a pipe in Nek5000.

Contributions

- A highly optimized version of Nekbone on GPUs inspired by Świrydowic et. al.
- We obtain close to the peak performance on one GPU based on the measured bandwidth of Nvidia P100 and V100 GPUs.

Nekbone

- Proxy app for the larger CFD solver Nek5000 based on the Spectral Element Method. Improvements made in Nekbone can easily be ported to the original solver.
- Solves the Poission equation with Dirichlet boundary conditions

$$\begin{aligned} -\nabla^2 u &= f, \qquad u \in \Omega \\ u &= 0, \qquad u \in \partial\Omega \end{aligned}$$

  on a cubic domain.
- Discretized with the spectral element method and the linear system is solved with the Conjugate Gradient method. Evaluation of the linear system is composed of two parts.
  - The local computation of the Poisson operator on each element, Ax.
  - The communication along the boundary between elements, *gather-scatter*.

Local Poisson operator

Expressed as small tensor operation between the differential matrix D, geometric factors G and the cubic element nodal values u. The small matrix-multipies are typically of size $6^3 - 12^3$ and libraries such as cuBLAS do not yield high performance. Expressed here as Fortran pseudocode.

```
do e = 1,nelt !Loop over all elements
   do i,j,k = 1,n+1 !Loop over i,j then k
      wr, ws, wt = 0
      do l=1,n+1
         wr = wr + D(i,l)*u(l,j,k,e)
         ws = ws + D(j,l)*u(i,l,k,e)
         wt = wt + D(k,l)*u(i,j,l,e)

      ur(i,j,k,e) = G(i,j,k,1,e)*wr
                  + G(i,j,k,2,e)*ws
                  + G(i,j,k,3,e)*wt
      us(i,j,k,e) = G(i,j,k,2,e)*wr
                  + G(i,j,k,4,e)*ws
                  + G(i,j,k,5,e)*wt
      ut(i,j,k,e) = G(i,j,k,3,e)*wr
                  + G(i,j,k,5,e)*ws
                  + G(i,j,k,6,e)*wt
do e = 1,nelt
   do i,j,k = 1,n+1 !Loop over i,j then k
      w(i,j,k,e) = 0.0
      do l=1,n+1
         w(i,j,k,e) = w(i,j,k,e)
         + D(l,i)*ur(l,j,k,e)
         + D(l,j)*us(i,l,k,e)
         + D(l,k)*ut(i,j,l,e)
```

Optimization

Baseline OpenACC and CUDA Fortran Implementation

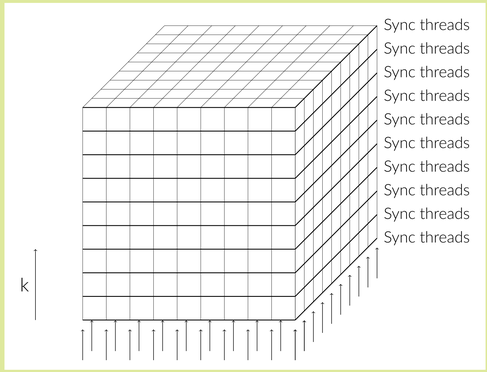
- We start from an inital OpenACC port of the Local Poisson operator and a corresponding CUDA Fortran implementation.
- Part of initial work by Jing Gong et. al. [1] and closely follows the structure from the pseudocode.
- Stores arrays u, D ang G in global memory, leads to a lot of data movement to and from global memory.

Leveraging Shared Memory

- Work by Joksch et. al [2], uses a 3D thread block to compute an element.
- Stores arrays in shared memory, enables faster bandwidth and fewer accesses to global memory.
- Limited in size, large elements do not fit in shared memory.

2D thread structure

- Our final optimization in Nekbone. Based on previous work by Świrydowic et. al [3] and we use it in the context of Nekbone.
- Computes each element "layer by layer". Each thread goes through a "pile" of nodal points in sync. Decreases shared memory usage, we do not run out of memory.
- Enables each thread to store its pile into registers → Increased temporal and spatial locality and faster memory bandwidth.
- Trades inexpensive thread synchronization over the thread block `__syncThreads()` for better locality.



The arrows illustrates how each thread propagates upwards through the element, enabling each thread to only work on its specific pile of points and compute each layer in sync. Compared to a 3D structure we trade increased locality for inexpensive synchronization over each thread block.

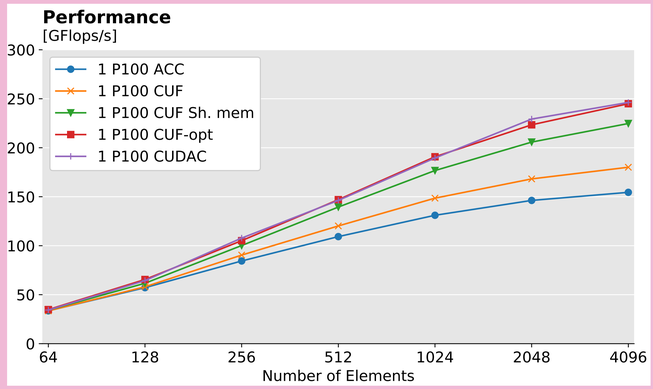
CUDA Fortran and CUDA C

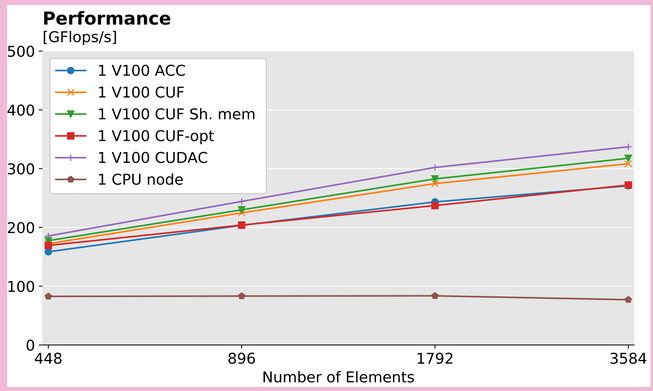
- CUDA Fortran lacks certain functionality, `#pragma unroll` and `__restrict` only in CUDAC.
- Compiler versions important for CUDA Fortran. Only PGI compiler supports interfacing Fortran with OpenACC and CUDA.

Experimental setup

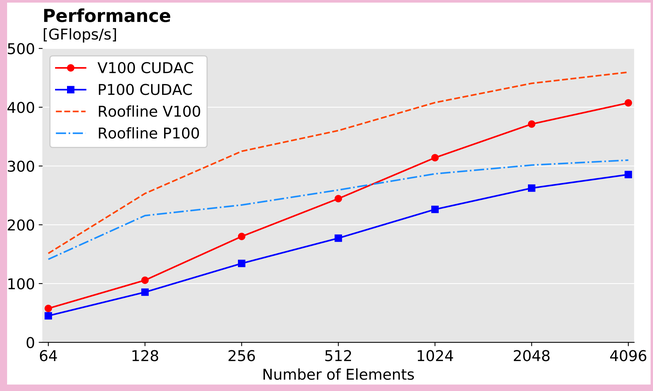
- Piz Daint @ CSCS equipped with P100 GPUs
- Kebnekaise @ HPC2N equipped with Nvidia V100 GPUs and Intel Xeon Gold 6132 CPUs with 28 cores at 2.6GHz.
- PGI Compiler and CUDA (PGI v. 19.7 and CUDA 10.1 on Piz Daint, v. 18.7 and 9.2 on Kebnekaise).
- Experiments were made on elements of degree 9 with 10^3 nodal points.
- Bandwidth measurements were made with `cudaMemcpyDeviceToDevice` for different array lengths.

Numerical Results





Performance for polynomial degree 9, element size 10^3 of the different GPU versions for a P100 GPU (Top) and a V100 GPU (Bottom) as well as one single CPU node (Bottom). The number of elements for the V100 were chosen as to match running a 28 core CPU with $32 - 128$ elements per core. We attribute the more modest performance improvement for the V100 to it's automatic usage of shared memory as a form of cache, making the benefit of explicitly using shared memory less pronounced.



Performance of the optimized CUDAC version with the gather-scatter (dssum) kernel disabled. This lets us compare the performance to an empirically measured roofline for our implementation based on the operational intensity and the measured bandwidth for different input sizes. From 1024 elements we obtain $80 - 90\%$ of peak performance.

Future Work

- Evaluate how different preconditioners can be optimized for GPUs and which preconditioners make good GPU candidates.
- Measuring the achieved performance in Nek5000 and for different polynomial degrees.
- Assessing whether the algorithm and especially the OpenACC pragmas can be improved.
- Investigating the communication kernels as to improve strong and weak scaling.

References

[1] Jing Gong, Stefano Markidis, Erwin Laure, Matthew Otten, Paul Fischer, and Misun Min. Nekbone performance on GPUs with OpenACC and CUDA fortran implementations. *J. Supercomput.*, 72, 07 2016.

[2] Andreas Jocksch, Jing Gong, Niclas Jansson, Adam Peplinski, Alan Gray, and Philipp Schlatter. Porting nek5000 on GPUs using OpenACC and CUDA. Poster at PASC'20, in press.

[3] Kasia Świrydowicz, Noel Chalmers, Ali Karakus, and Tim Warburton. Acceleration of tensor-product operations for high-order finite element methods. *Int. J. High Perform. Comput. Appl.*, 33(4):735--757, 2019.

This work was supported by a European Commission Horizon2020 project grant entitled "EXCELLERAT: The European Centre of Excellence for Engineering Applications" (grant reference 823691). Financial support from the SeRC Exascale Simulation Software Initiative (SESSI) is gratefully acknowledged. The computations were performed on resources provided by the PRACE Research Infrastructure resource PizDaint hosted by CSCS, Switzerland and the Swedish National Infrastructure for Computing (SNIC) at High Performance Computing Center North (HPC2N).

Authors' emails: [makarp, njansson, podobas, markidis] @kth.se and pschlatt@mech.kth.se.