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
Stockholm, Sweden
makarp@kth.se, njansson@kth.se, podobas@kth.se, pschlatt@mech.kth.se, markidis@kth.se

***Abstract*—In the CFD solver Nek5000, the computation is dominated by the evaluation of small tensor operations. Nekbone is a proxy app for Nek5000 and has previously been ported to GPUs with a mixed OpenACC and CUDA approach. In this work, we continue this effort and optimize the main tensor-product operation in Nekbone further. Our optimization is done in CUDA and uses a different, 2D, thread structure to make the computations layer by layer. This enables us to use loop unrolling as well as utilize registers and shared memory efficiently. Our implementation is then compared on both the Pascal and Volta GPU architectures to previous GPU versions of Nekbone as well as a measured roofline. The results show that our implementation outperforms previous GPU Nekbone implementations by 6-10%. Compared to the measured roofline, we obtain 77 - 92% of the peak performance for both Nvidia P100 and V100 GPUs for inputs with 1024–4096 elements and polynomial degree 9.**

***Index Terms*—Nekbone, Nek5000, CUDA, OpenACC, GPU**

I. Introduction to Nekbone and Ax

Nek5000 [1] is a high order solver for Computational Fluid Dynamics (CFD) based on the Spectral Element Method (SEM) that integrates the incompressible Navier-Stokes equation in time. Currently, the solver uses CPUs and Fortran 77 combined with C and MPI for parallelization. As we are moving to exa-scale computations and more heterogeneous hardware [2] a major effort to modernize Nek5000 and port it to GPUs and other accelerators is necessary. Nekbone [3] is a proxy app for Nek5000 that illustrates important computational and scaling aspects of the entire solver. In particular, the evaluation of the Poisson operator through a tensor-product operation is the most time consuming part of both Nekbone and Nek5000 [4]. In this work, we examine how the tensor-product operations in Nekbone can be optimized for two state-of-the-art GPU architectures: Pascal and Volta [5]–[7]. In addition, we evaluate the optimization compared to a measured roofline [8], [9] to assess how close to peak performance we are.

Our contribution is to utilize newly developed optimization techniques for GPUs and higher order finite element methods in Nekbone. To do this we use a mixed CUDA and OpenACC approach to interface with the legacy Fortran codebase. The resulting GPU version of Nekbone is the highest performing yet.

Nekbone isolates the core computational components for Nek5000 by discretizing the Poisson equation with SEM on a deformed cubic domain. It then uses the Conjugate Gradient (CG) method to solve the resulting linear system $Ax = f$. We compute the the result of the Ax operation in two steps. First, we express the local Poisson operator on each element as a tensor-product operation. Then, in the second step, we communicate the local computation results to the neighboring elements. We focus on optimizing the local tensor-products performed on each element. The original approach is illustrated in Listing 1. In the pseudocode, we compute the local evaluation, `w`, of the Poisson operator for each element as a function of the nodal values `u`, the differential matrix `dxtm1` and the geometric factors `gxyz` similarly to how it is expressed in Nekbone. Note that the main operations are in essence small matrix multiplications and the operational intensity is $O(n+1)$ where n is the order of our polynomial approximation and $n+1$ the number of collocation points.

Listing 1
PSEUDOCODE FOR THE TENSOR-PRODUCT USED TO EVALUATE THE POISSON OPERATOR IN NEKBONE.

```
do e = 1,nelt !Loop over all elements
   do i,j,k = 1,n+1 !Loop over i,j then k
      wr = 0
      ws = 0
      wt = 0
      do l=1,n+1
         wr = wr + dxm1(i,l)*u(l,j,k,e)
         ws = ws + dxm1(j,l)*u(i,l,k,e)
         wt = wt + dxm1(k,l)*u(i,j,l,e)
      enddo
      ur(i,j,k,e) = gxyz(i,j,k,1,e)*wr
                  + gxyz(i,j,k,2,e)*ws
                  + gxyz(i,j,k,3,e)*wt
      us(i,j,k,e) = gxyz(i,j,k,2,e)*wr
                  + gxyz(i,j,k,4,e)*ws
                  + gxyz(i,j,k,5,e)*wt
      ut(i,j,k,e) = gxyz(i,j,k,3,e)*wr
                  + gxyz(i,j,k,5,e)*ws
                  + gxyz(i,j,k,6,e)*wt
do e = 1,nelt  !Loop over all elements
   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)
         + dxtm1(i,l)*ur(l,j,k,e)
         + dxtm1(j,l)*us(i,l,k,e)
         + dxtm1(k,l)*ut(i,j,l,e)
```

II. Nekbone on GPUs

In the GPU implementation of Nekbone, the PGI compiler is used to compile the code on GPUs with a mixed OpenACC and CUDA approach [10].

A. Original Implementation

In a original Nekbone implementation for GPUs by Gong et al. [10], the Ax evaluation is implemented similarly to the pseudocode in Listing 1. Their original version suffers from a poor temporal locality, and only global memory is used. This initial approach has both a CUDA Fortran [11] and OpenACC implementation. In this subroutine, they allocate one thread block per element and then utilize as many threads as possible to

compute the element with some stride (depending on the number of threads and size of the element). Despite the greedy use of threads for each element, they are not organized for locality.

B. Leveraging shared memory

Previous work focused on optimizing the kernel to exploit GPU shared memory [12] and achieved this by loading all the nodal values and the differential matrix into GPU shared memory. However, since they utilized the same thread structure as in the original approach they needed to load the entire element into shared memory. A problem with this is the limitation in GPU shared memory capacity and it prevents storage of the whole element for larger element sizes.

C. 2D Thread Structure

Our main optimization is to use a 2D thread structure when creating our thread blocks. This thread structure avoids the problems in the shared memory version by only having a "layer" of points in shared memory at a time. The use of this thread structure was studied in [9] and it enables other optimizations such as loop unrolling. What this 2D thread structure means is that instead of allocating as many threads as possible per element, each thread block uses a layer of $n \times n$ threads. By reorganizing the threads in this way, the outer two element-level loops in Listing 1, as well as the one iterating over k, can be merged. This means that each thread block goes through each k layer in lock-step, enabling the threads to save `u, w` into registers as well as preloading the geometric factors, in addition to having `dxtm1` in GPU shared memory. When neighboring nodal values are needed for the second part of the derivative, the threads in the block are synchronized (`__syncThreads()`). We made this implementation in CUDA C and CUDA Fortran. When using CUDA C, loop unrolling (`#pragma unroll`) as well as marking memory as read-only was used for more compiler-side optimizations. For our optimized CUDA Fortran version, the loop was manually unrolled once instead.

III. Numerical Experiments

We perform experiments on Piz Daint at CSCS in Switzerland and Kebnekaise at HPC2N in Sweden. Piz Daint is equipped with Cray XC50 compute nodes and each node has a 12 core Intel Xeon E5-2690 v3 @ 2.6GHz and one Nvidia Tesla P100 GPU. Kebnekaise has a Intel Xeon Gold 6132 with 28 cores @ 2.6GHz and two Nvidia V100 GPUs per node. We use the PGI compiler 19.7 and CUDA version 10.1 on Piz Daint while on Kebnekaise we use version 18.7 and 9.2. In all measurements, we run Nekbone with 100 CG iterations, $n = 9$.

We measure the performance of the original CUDA Fortran version of the Poisson operator, the OpenACC version, the shared memory version by Gong et al., as well as our optimized CUDA Fortran and CUDA C implementations. In Fig. 1 we can observe the performance of all the major versions of Nekbone on a Nvidia P100 GPU while in Fig. 2 the performance is shown for a Nvidia V100. The restructuring of the kernel into 2D thread blocks gives improved performance. In particular, the difference between using global, shared, or register memory is noticeable. However, for the measurements on Nvidia V100 GPU, we do not observe any performance gain for the optimized CUDA Fortran kernel, but rather a slowdown. On Piz Daint, with a later version of the PGI compiler, the difference between our optimized CUDA C and CUDA Fortran kernels is less than 1%. We include the CPU performance on Kebnekaise to illustrate the significant dependence on the problem size of the GPU version.

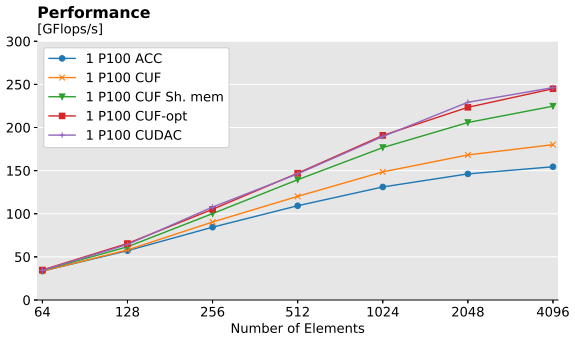

Fig. 1. Performance for different versions of Nekbone on Piz Daint with one Nvidia P100 GPU.

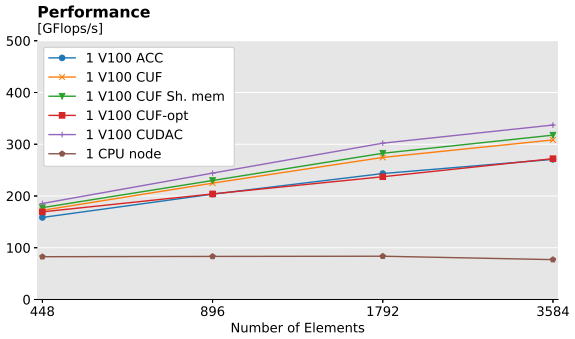

Fig. 2. Performance for different versions of Nekbone on Kebnekaise. The CPU version is run on one node with 28 cores and MPI for parallelization. The GPU measurements use one Nvidia V100 GPU.

To evaluate how close we are to peak performance, we construct a measured roofline [8], [9] in Fig. 3. This plot implies that we are in the memory-bound domain [13]. Based on measurements of the bandwidth and our operational intensity, we note that we are close to the peak performance of the GPU. For 1024, 2048, 4096 elements we achieve 78%, 87%, 92% of the roofline for the P100 and 77%, 84%,88% for the V100.

References

[1] P. F. Fischer, J. W. Lottes, and S. G. Kerkemeier. Nek5000 website. Accessed: May 21, 2020. [Online]. Available: https://nek5000.mcs.anl.gov/

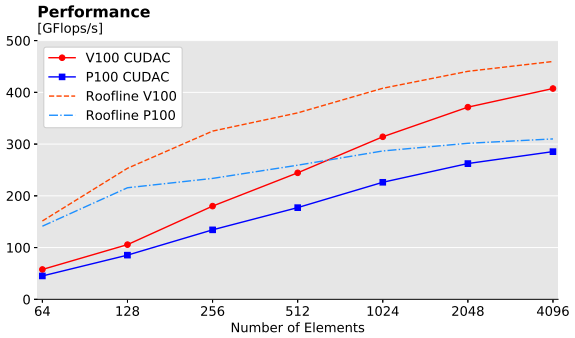

Fig. 3. Measured roofline for the P100 GPU on Piz Daint as well as the same measurements performed on a V100 GPU on Kebnekaise.

[2] Top 500 List. Accessed: May 21, 2020. [Online]. Available: https://www.top500.org/

[3] F. Sheet, "Collaboration of Oak Ridge, Argonne, and Livermore (CORAL)," *US Department Of Energy, National Nuclear Security Administration*, 2014.

[4] S. Markidis *et al.*, "OpenACC acceleration of the nek5000 spectral element code," *Int. J. High Perform. Comput. Appl.*, vol. 29, no. 3, pp. 311–319, 2015.

[5] "Nvidia tesla P100," Nvidia, Santa Clara, United States, White paper, accessed: May 21, 2020. [Online]. Available: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[6] "Nvidia tesla V100 GPU architecture," Nvidia, Santa Clara, United States, White paper, August 2017, accessed: May 21, 2020. [Online]. Available: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[7] S. Markidis, S. W. D. Chien, E. Laure, I. B. Peng, and J. S. Vetter, "Nvidia tensor core programmability, performance & precision," in *IPDPSW*. IEEE, 2018, pp. 522–531.

[8] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[9] K. Świrydowicz, N. Chalmers, A. Karakus, and T. Warburton, "Acceleration of tensor-product operations for high-order finite element methods," *Int. J. High Perform. Comput. Appl.*, vol. 33, no. 4, pp. 735–757, 2019. [Online]. Available: https://doi.org/10.1177/1094342018816368

[10] J. Gong *et al.*, "Nekbone performance on GPUs with OpenACC and CUDA fortran implementations," *J. Supercomput.*, vol. 72, 07 2016.

[11] *CUDA Fortran Programming Guide and Reference*, PGI Compilers and Tools, Beaverton, United States, 2017, accessed: May 23 2020. [Online]. Available: https://www.pgroup.com/doc/pgi17cudaforug.pdf

[12] A. Jocksch *et al.*, "Porting nek5000 on GPUs using OpenACC and CUDA," Poster at PASC'20, in press.

[13] J. Domke *et al.*, "Double-precision fpus in high-performance computing: an embarrassment of riches?" in *IPDPS*. IEEE, 2019, pp. 78–88.