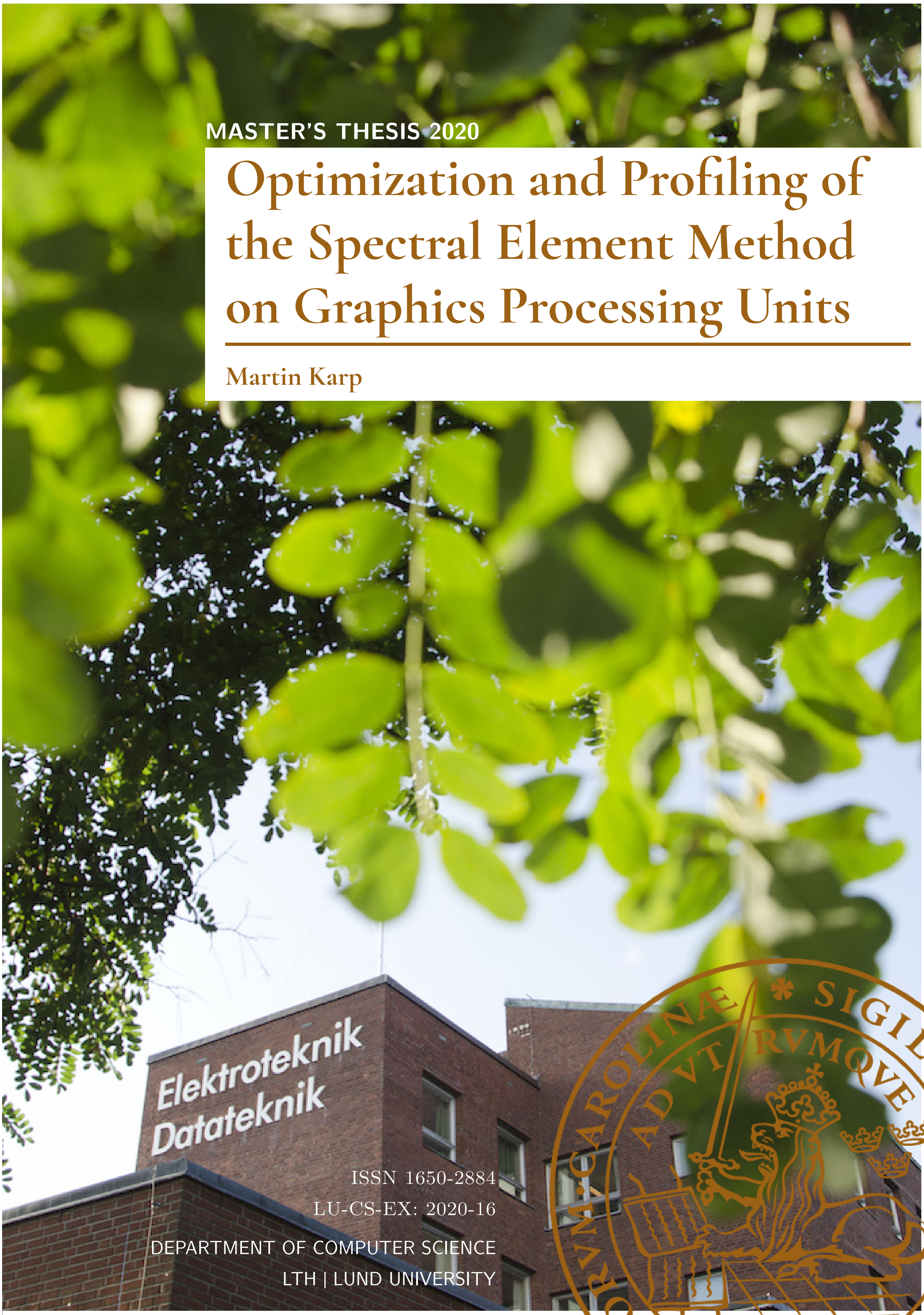
MASTER'S THESIS 2020

Optimization and Profiling of the Spectral Element Method on Graphics Processing Units

Martin Karp



ISSN 1650-2884
LU-CS-EX: 2020-16

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2020-16

Optimization and Profiling of the Spectral Element Method on Graphics Processing Units

Martin Karp

Optimization and Profiling of the Spectral Element Method on Graphics Processing Units

Martin Karp
martin.karp.073@student.lu.se

May 20, 2020

Master's thesis work carried out at the PDC Center for High Performance Computing at KTH.

Supervisors: Dr. Niclas Jansson, njansson@kth.se
Dr. Jonas Skeppstedt, jonas.skeppstedt@cs.lth.se

Examiner: Prof. Krzysztof Kuchcinski, krzysztof.kuchcinski@cs.lth.se

Abstract

In this work, optimizations of the Spectral Element Solver Nek5000 on single and multiple GPUs were investigated. In particular, in the proxy app Nekbone, the main computation was optimized for a single GPU with a mixed OpenACC and CUDA approach. Additionally, measurements were taken to asses the scaling of the code for multiple GPUs. The results showed that a close to optimal single GPU implementation of Nekbone had been achieved. For future performance increases however, it was concluded that the current communication kernel is the main bottleneck for scaling on multiple GPUs.

Keywords: Nek5000, Nekbone, CUDA, GPU, OpenACC, Roofline, SEM

Acknowledgements

First, I thank my supervisors Jonas and Niclas for their support, expertise and great feedback during my thesis..

I would like to thank all my friends with whom I have shared many great memories during my studies. I direct a thanks to my family, you are my safe harbor. And thank you, Olivia, for having been there almost to the end.

Without anyone of you, this thesis and these past five years would not have become what they became.

Contents

Chapter 1
Introduction

Nek5000 is a scientific solver for Computational Fluid Dynamics (CFD) based on the Spectral Element Method (SEM) that is highly scalable and used extensively in academia. One important and the most computationally expensive aspect of the solver is solving the Poission equation to calculate the pressure. This part of the solver is solved in the Proxy app Nekbone to illustrate the scalability and performance of the whole Nek5000 solver. In Nekbone the Poission equation is solved with the Conjugate Gradient Method with an optional multigrid preconditioner and has successfully scaled to millions of MPI ranks. However, both Nek5000 and Nekbone have only to a limited extent been ported to multiple Graphics processing units (GPUs). Using multiple GPUs could potentially increase the performance of the solver by a large margin and this work aims to illustrate the possible performance gains in Nek5000 by optimizing the important math operations in Nekbone on GPUs.

1.1 Problem statement

A large issue with older numerical codes is portability and modifying the codebase to fit new hardware such as GPUs. In recent times with the advent of machine learning and therefore also an increased number of GPU clusters, the effort to utilize GPUs for other numerical calculations has increased tremendously. In this report we discuss the performance of Nekbone on single and multiple GPUs. In particular we look at optimizing the performance of Nekbone on a single GPU and use previous work by Jing et. al. [4] to evaluate the bottlenecks when scaling to multiple GPUs.

The research questions are as follows,

- What performance can be achieved and is currently achieved in Nekbone on single and multiple GPUs?
- What performance limiting factors are there for Nekbone?

1.2 Previous work

In the area of utilizing GPUs for scientific computing, work has been done to some extent since the early 2000s by using shaders and other constructs made for graphical applications. However, with the advent of CUDA and OpenCL it has been possible to utilize GPUs for general-purpose programming or so called GPGPUs [7]. Because of the inherent parallelism in the GPU architecture, compute-intensive applications such as matrix multiplication have successfully been able to utilize GPUs well. One such application is the Spectral Element Method and it is, therefore, an ongoing work to improve this method for GPUs and in particular multiple GPUs.

Previous work on Nekbone specifically has been done by J Gong et. al.[4] where the main conjugate gradient method was parallelized for multiple GPUs with OpenACC and CUDA Fortran. In their work, it was shown that using a simple pragma based approach such as OpenACC for simpler operations and then parallelizing the main computation subroutine, $\boldsymbol{Ax}$, a good trade-off between portability and performance could be achieved. However, it was also noted that the communication between different nodes and elements posed a major bottleneck for scaling the code on more GPUs. In a workshop following this, N Jansson, J Gong and A Peplinski tried to optimize the computation subroutine $\boldsymbol{Ax}$ further and improved the performance with improved use by shared memory and the utilization of CUDA C instead. Further work on optimizing $\boldsymbol{Ax}$ was also recently done in a paper by Świrydowicz et. al. [11], where they used the accelerator framework OCCA and managed to achieve performance at or very close to the maximal performance based on the bandwidth.

1.3 Contributions

In this work we combine the work by Gong and Świrydowicz et.al. to obtain a near-optimal single GPU implementation of Nekbone, based on the measured GPU memory bandwidth and roofline analysis. In addition to this, we investigate the performance impact of the current communication kernel in preparation for further improvements of Nek5000.

1.4 Delimitations

Certain assumptions were made to make it feasible to complete the project in the allotted time. In particular, we only focus on Nekbone as a way of illustrating possible changes in Nek5000. We never implement any optimizations in Nek5000. In addition to this, the work optimizes and investigates the performance of Nekbone without a preconditioner, the preconditioner is never optimized.

Chapter 2
Background

In this chapter, we present the mathematical theory behind Nek5000/Nekbone and the exact formulation of the algorithm. We will also cover the most important aspects of a GPU, especially the key differences compared to a conventional Central Processing Unit (CPU).

2.1 Nekbone

Nekbone aims to capture the main computational parts of the larger CFD code Nek5000. This is done by solving the Poisson equation with Dirichlet boundary conditions through the Spectral Element Method (SEM) on a cubic domain. Solving the Poisson equation is the most time-consuming part of Nek5000 and showcasing performance improvements in Nekbone can, therefore, showcase future improvements to the entire solver. We will now look into the mathematical background behind SEM and illustrate how the method works.

2.1.1 The Spectral Element Method

The Spectral Element Method is a combination of the Finite Element method with higher-order base functions. In other words, the method revolves around decomposing the computational domain into small elements and then solving the smaller problem on each element with a Spectral solver with high order polynomials as base functions. The solution along the boundary of each element is then communicated to the bordering elements. Using high order polynomials lead to a relatively high computational intensity per element which makes the method a good candidate for acceleration with GPUs.

As mentioned, we solve the Poisson equation with Dirichlet boundary conditions

$$\begin{aligned} -\nabla^2 u &= f, \qquad & u \in \Omega \\ u &= 0, \qquad & u \in \partial\Omega \end{aligned} \tag{2.1}$$

where Ω is the computational domain and $\partial\Omega$ is the domain boundary. In the case of Nekbone the domain is a cube, but the domain Ω can take many shapes. The strong formulation is the most common, but for SEM we consider the weak formulation of (2.1) defined as:

Find $u \in H_0^1(\Omega)$ such that

$$(\nabla u, \nabla v) = (f, v) \qquad \forall v \in H_0^1(\Omega), \tag{2.2}$$

where

$$(u, v) = \int_\Omega u \cdot v d\Omega, \qquad \forall u, v \in L^2 \tag{2.3}$$

and the spaces are defined as $L^2 = \{v : \int_\Omega v d\Omega < \infty\}$ and $H_0^1 = \{v : v(\partial\Omega) = 0, v \in L^2, \nabla v \in L^2\}$ [6]. In addition, we used the identity that $(\nabla^2 u, v) = (\nabla u, \nabla v)$ in L^2 when rewriting to weak form [9]. Putting the formulation in words, we are looking for u that solves the Poisson equation in H_0^1, which is the square-integrable Sobolev space that is disappearing on the boundary $\partial\Omega$ [4]. However, since this is an infinite function space, we need to approximate the solution by containing it to a finite-dimensional subspace $X_0^N \subset H_0^1$. In SEM we do not look for the exact solution of u in H_1^0, but rather an approximation in the discrete space X_0^N[9].

Let us then introduce X_0^N. First off we need to choose base functions for each element. The ones used in Nekbone are the Nth Legendre polynomial, L_N, interpolated with Lagrange interpolation on the so-called Gauss-Lobatto-Legendre(GLL) points[4]. The GLL points and the one dimensional Legendre polynomials are closely related and defined on the interval $\omega = [\,-1, 1\,]$. The Legendre polynomials can be computed according to the recurrence [9]

$$\begin{aligned} &L_0(\xi) = 1, \qquad L_1(\xi) = \xi, \\ &L_k(\xi) = \frac{1}{k}((2k-1)\xi L_{k-1}(\xi) - (k-1)L_{k-2}(\xi)) \end{aligned} \tag{2.4}$$

and the GLL points are defined as the zeros of

$$(1 - \xi^2)L_N'(\xi), \tag{2.5}$$

which we will refer to as $\xi_i, i \in \{0, \dots, N\}$.

With this we can now form the one-dimensional base functions l_i in ω

$$l_i(\xi) = \frac{N(1-\xi^2)L_N'(\xi)}{(N+1)(\xi - \xi_i)L_N'(\xi_i)} \tag{2.6}$$

where ξ_i is the ith GLL point. This gives us a set of base functions l_i that are orthogonal in one dimension and where the relation $l_i(\xi_j) = 0, i \neq j$ holds. We now need to extend these base functions into three dimensions, which is done by taking the tensor product of the base functions to obtain $\psi_{ijk}(\xi, \mu, \gamma) = l_i(\xi)l_j(\mu)l_k(\gamma)$ where $(\xi, \mu, \gamma) \in \omega^3$ [6]. When we now have a base in this normalized space, we have connect to the problem on our original domain Ω.

This is done by making a non-overlapping partition of $\Omega = \bigcup_e^E \Omega^e$ where each element Ω^e and its coordinates $(x_1, x_2, x_3) = \mathbf{x} \in \Omega^e$ are mapped to a corresponding set of local coordinates $r(\mathbf{x}) = (r_1, r_2, r_3) \in \omega^3$ [4]. By now taking the intersection $H_0^1 \cap V = \text{span}(\psi_{ijk})$ we obtain X_0^N. Each function on each element can then be expressed as [9]

$$u^e(x_1, x_2, x_3) = \sum_{i=0}^{N} \sum_{j=0}^{N} \sum_{k=0}^{N} u^e_{ijk} \psi_{ijk}(r_1, r_2, r_3) = \sum_{i,j,k=0}^{N} u^e_{ijk} \psi_{ijk}. \tag{2.7}$$

With this we can now define scalar product and differentiation in X_0^N by reformulating the weak problem statement into discrete form. First, however we need to define scalar product and differentiation in X_0^N. Scalar product, $(u, v)_N$ is defined as [9]

$$(u, v)_N = \sum_{e=1}^{E} \sum_{i,j,k=0}^{N} u^e_{ijk} \psi_{ijk} v^e_{ijk} \psi_{ijk}. \tag{2.8}$$

Next we need to evaluate the $\nabla = (\frac{\partial}{\partial x_1}, \frac{\partial}{\partial x_2}, \frac{\partial}{\partial x_3})$ operator. Following from equation (2.7) we get

$$\frac{\partial u}{\partial x_i} = \sum_{j=1}^{3} \frac{\partial u}{\partial r_j} \frac{\partial r_j}{\partial x_i}. \tag{2.9}$$

Now, we can express $(\nabla u, \nabla v)_N$ as

$$(\nabla u, \nabla v)_N = \sum_{e=1}^{E} \sum_{i=1}^{3} \sum_{j=1}^{3} \iiint_{\omega^3} \frac{\partial u}{\partial r_i} \mathcal{G}_{ij} \frac{\partial v}{\partial r_j} dr_1 dr_2 dr_3 \tag{2.10}$$

where we have introduced the geometric factors $\mathcal{G}_{ij}$ as

$$\mathcal{G}_{ij} = \sum_{k=1}^{3} \frac{\partial r_i}{\partial x_k} \frac{\partial r_j}{\partial x_k} \left| \frac{\partial \mathbf{x}}{\partial \mathbf{r}} \right| \tag{2.11}$$

which are essentially coefficients dependant on the derivatives of the global position x relative to the local position in the element $\mathbf{r}$ [4]. These factors are therefore also constant for a constant geometry. As for the differentials of u and v, they also only depend on the base functions and its own weight according to

$$\frac{\partial u}{\partial r_l} = \sum_{i,j,k=0}^{N} u^e_{ijk} \frac{\partial \psi_{ijk}(r_1, r_2, r_3)}{\partial r_l}. \tag{2.12}$$

With these definitions we now have our final problem statement: find $u \in X_0^N$ such that

$$(\nabla u, \nabla v)_N = (u, v)_N, \qquad \forall v \in X_0^N. \tag{2.13}$$

Discretization

To numerically solve the SEM system we need to rewrite the equations on matrix form and introduce a global numbering. The alert reader may have noticed that we have so far not spent any time explaining how we deal with shared nodal points between the elements and how continuity is enforced. We start by introducing the global weight vectors $\underline{u}$ and $\underline{v}$ which are the global value vectors containing every unique weight u^e_{ijk}, i.e. we only have one value for each position $\mathbf{x} \in \Omega$ [6]. To maintain that each weight is mapped to its corresponding element we, therefore, need to introduce the gather-scatter matrix Q which is the global to local map of $\underline{u}$. In other words, Q is a boolean matrix where a 1 at index i, j in Q indicates that the weight at index j in $\underline{u}$ belongs in the ith element [6].

For the boundary conditions though, we must introduce another matrix as well, the so called restriction matrix R which zeros our any weight along the boundary. It is similar to Q, but only zeros out the values along the boundary.

The next matrix we need is the differentiation matrix. If we look at (2.12) we see that the partial derivative of ψ is reused in every element together with the geometric derivative $\mathcal{G}$. By pre-computing these matrices and approximating the integral in (2.10) with the quadrature over the GLL nodal points we have two constant matrices that only depend on the geometry and the base functions. Starting with the differentiation of u, v, we get the differential matrix

$$D = \begin{pmatrix} D_\xi \\ D_\mu \\ D_\gamma \end{pmatrix} = \begin{pmatrix} I \otimes I \otimes \hat{D} \\ I \otimes \hat{D} \otimes I \\ \hat{D} \otimes I \otimes I \end{pmatrix} \tag{2.14}$$

where $\hat{D}_{ik} = l'_k(\xi_i)$. As for the matrix with the geometric factors, we get a symmetrical matrix

$$G^e = \begin{pmatrix} G^{11} & G^{12} & G^{13} \\ G^{21} & G^{22} & G^{23} \\ G^{31} & G^{32} & G^{33} \end{pmatrix} \tag{2.15}$$

with values $G^{lm}_{ijk} = \mathcal{G}^{lm}_{ijk}\rho_i\rho_j\rho_k$. The last matrix we need for our system is for simply computing the scalar product (v, f) which is the diagonal mass matrix B^e which is defined for each element as $B^e_{ijk} = \left|\frac{\partial \mathbf{x}}{\partial \mathbf{r}}\right|\rho_i\rho_j\rho_k$ i.e. the jacobian evaluated and multiplied with the corresponding GLL weights at local index i, j, k [4].

Now we can assemble our system. To make use of the restriction and gather-scatter matrix R, Q we need to assemble the local matrices into global matrices. This is done by taking the block-diagonal of all the local element matrices and assembling them into one. This is simply done by defining the global matrices $G = G^1 \oplus G^2 \ldots \oplus G^E$ and repeating for the other matrices. If this notation is unfamiliar, the result is a matrix with each element matrix as a block along the diagonal.

With this, we can rewrite our system in (2.13) as

$$\underline{v}^T R Q^T D^T G D Q R^T \underline{u} = \underline{v}^T R Q^T B f \tag{2.16}$$

which after rewriting and simplifying gives us

$$\begin{aligned} A\underline{u} &= RQ^T Bf \\ A &= RQ^T D^T G D Q R^T. \end{aligned} \tag{2.17}$$

In other words it all comes down to solving a very large, sparse linear system of equations. It should be pointed out that the matrix A is never explicitly formed and that actually constructing A would also be too expensive to be feasible. Everything is rather computed for each element and then communicated in a gather-scatter operation that performs the action of Q. The level of parallelism is therefore very high since each element can be operated on independently except for the communication along the boundary.

Now that we have formalized our problem into a linear system of equations we move on to how to actually solve the linear system.

2.1.2 The Conjugate Gradient Method

A large issue with conventional explicit solvers for linear systems $Ax = f$ is that the inverse of the matrix A becomes too computationally expensive to compute. That is why for systems dealing with millions of degrees of freedom, as in our case, iterative or Krylov methods are used [5]. Another issue in our case is that we never actually form A at all, but this is also avoided by using Krylov methods for which we only need to compute the product Ax. They are not dependent on the actual formation of A.

In the case of Nekbone, the Conjugate Gradient (CG) method specifically is used to solve the system $Ax = f$ [4]. The general idea of the CG method is to approximate the solution as a composition of base vectors v_i and minimize the function $r_i = kAv - r_{i-1}, \quad k \in R, r_0 = f$[5] at each step i. Then, the approximate solution of x at iteration i can be written as $x_i = x_{i-1} + kv_n$. This is then repeated until the residual r_i is below some threshold. To improve the rate of convergence, a preconditioner can be applied to the system, M. In pseudocode the algorithm is defined as in algorithm 1 below. In the pseudocode the procedure is explicitly stated in matrix form, but it corresponds well with how it is formulated in Nekbone. In this work we will focus on the method without the preconditioner, i.e. $M = I$.

Algorithm 1 Pseudocode for the Conjugate Gradient Method

```
procedure Conjugate Gradient Method
Input: $A \in R^{n \times n}, \mathbf{f} \in R^n$
Output: $\mathbf{x} \in R^n$
    $\mathbf{r}_0 \leftarrow \mathbf{f}, \mathbf{x} \leftarrow \mathbf{0}$
    $\mathbf{p}_0 \leftarrow \mathbf{r}_0$
    while $||r_i||_2 > \varepsilon$ do
        $\mathbf{z}_i \leftarrow M^{-1}\mathbf{r}_i$
        $\beta_i \leftarrow \frac{\mathbf{r}_i^T \mathbf{z}_i}{\mathbf{r}_{i-1}^T \mathbf{z}_{i-1}}$
        $\mathbf{p}_i \leftarrow \mathbf{z}_i + \beta_i \mathbf{p}_{i-1}$
        $\mathbf{w}_i \leftarrow A\mathbf{p}_i$
        $\alpha_i \leftarrow \frac{\mathbf{r}_i^T \mathbf{z}_i}{\mathbf{p}_i^T \mathbf{w}_i}$
        $\mathbf{x}_i \leftarrow \mathbf{x}_{i-1} + \alpha_i \mathbf{p}_i$
        $\mathbf{r}_i \leftarrow \mathbf{r}_i - \alpha_i \mathbf{w}_i$
        $i \leftarrow i + 1$
    end while
end procedure
```

2.1.3 Cost analysis

To asses the performance of Nekbone we need to analyze the number of operations performed in each iteration in Nekbone and make an analysis of the data movement involved. As for the analysis we have two factors that contribute to the total number of degrees of freedom n. These are the number of elements E as well as the number of GLL points, N, required for the polynomial approximation. From this we get that $n = EN^3$. N is therefore always one higher than the degree of the polynomial approximation.

In Nekbone, the only large scale operations that are performed are adds and mults and we will therefore not make any detailed analysis of which Floating Point Operation (flop) that is performed. We will in other words not distinguish between the difference between a mult and an add even-though their latency and throughput differs slightly.

Let us now consider the cost of each iteration of Nekbone. The cost analysis of the CG method, if we do not consider the matrix multiplication of A and do not utilize a preconditioner, is not very costly. It is mostly vector additions and subtractions that are made and the total number of flops performed are

$$C_{CG}(n) = 15n. \tag{2.18}$$

As for the Matrix multiplication $A\mathbf{x}$, the number of computations, in this case, will be higher since the number of operations also scale with the number of nodal points N. If we consider the matrices from section 2.1.1 and count the number of operations we get

$$C_{SEM}(N, n) = n(12N + 19). \tag{2.19}$$

The total cost then becomes

$$C(N, n) = n(12N + 34) \tag{2.20}$$

for each iteration in Nekbone. Another factor that we need to consider is also the data transfers and the computational intensity of the code. Since we will port it to GPU, with higher computational intensity, we hope to achieve higher performance.

As for the number of reads and writes we have a total of $24n$ loads and $6n$ stores in Nekbone and therefore a total of $30n$ load/store operations. The total amount of memory per iteration is therefore $8 \cdot 30n$ since all computations are made in double precision.

Combining these two measures we can calculate the computational intensity T [flops/byte] as

$$T(N) = \frac{n(12N + 34}{8 \cdot 30n} = \frac{12N + 34}{240} \tag{2.21}$$

and we can, therefore, expect our performance on GPU to increase as the polynomial order is increased.

2.2 Acceleration on Graphics Processing Units

GPUs are increasingly popular in scientific computing as accelerators for certain applications. Nekbone and SEM are contenders for utilizing GPUs well because of the relatively high computational intensity compared to finite volume methods in CFD [7]. However, even

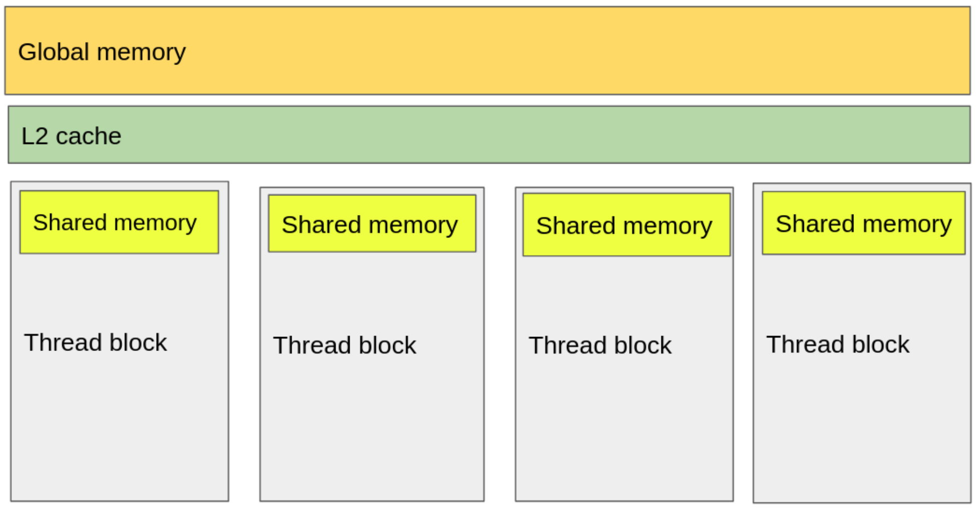


Figure 2.1: A greatly simplified scheme over a GPU architecture. In particular we show how all thread blocks have access to L2 cache and global memory while the threads in a thread block have one shared memory.

if the polynomial degree increases the computational intensity linearly, most applications use N between 6 and 10 as a trade-off between accuracy and performance [8]. The trade-off being that having more points leads to a finer grid and a higher rate of convergence, but it also makes the computation take longer time because of the CFL condition, which relates the granularity of the grid to the time step [1]. Another important issue is the problem of scalability. Even if one GPU can greatly improve performance compared to a CPU, the communication overhead between multiple GPUs can make a significant performance impact. In this section we will present the architecture of GPUs and introduce the roofline model as a way of measuring achievable performance for GPUs.

2.2.1 GPU architecture

A GPU differs from a CPU in that instead of having a few high performing threads, it has several hundreds of cores and processes several thousands of simpler threads at the same time. The focus is therefore on parallelism. All of these threads share a large global memory, but have their own individual registers. An important aspect of General Purpose GPU computing is to eliminate as much data movement between CPU and GPU as possible since the transfer speed between the two is comparably slow. A simplified picture of a GPUs architecture is provided in Figure 2.1. We will now go through the parts and functions of a GPU that will be relevant in the future discussion.

- **Kernel**, a function executed on the GPU. In general, it is a function executed by multiple threads that operate on small amounts of data in parallel.

- **Thread**, exactly like a thread on a CPU. Each thread executes the kernel sequentially and has its own registers.

- **Warp/Thread block**, is made up of several threads that share certain resources, illustrated in Figure 2.1, and can perform certain operations independently from other blocks,
  - **Shared memory**, is shared among all threads in a thread block and is faster than global memory but slower than registers.
  - **Syncing**, a warp can sync its own threads without stopping all other warps when executing a kernel.
- **Global memory**, shared among all thread blocks and is similar to Random Access Memory(RAM) for CPUs. It is slower than shared memory and registers, but faster than RAM on a CPU. Shown in Figure 2.1.
- **L2 cache**, global shared cache among all threads. It is usually not very large, but for smaller inputs it can make a large impact on read/write speed compared to global memory.

2.2.2 The Roofline Model

The roofline model has for a long time been used to assess what performances can be achieved depending on the computational intensity of a program on a specific CPU [10]. Because the bandwidth for the CPU and each level of cache and memory is limited, the peak performance of a CPU is often not achievable. For GPUs a similar approach can be taken where one takes into account the bandwidth of each level of memory. In our case we will focus on the Global memory, similarly to Świrydowicz et. al. [11].

The idea of the roofline model then is that the maximal performance of a CPU is not always achievable because of limited bandwidth. This means that the computational intensity combined with the bandwidth B of some level of memory can be used to assess the maximal performance can be achieved according to

$$\text{Roofline} = \min\{T \cdot B, P\} \tag{2.22}$$

where T is the computational intensity and P theoretical peak performance of the computing unit. The roofline model is therefore a simple and practical for rough estimates, but to consider how close one is to reaching maximal performance it poses some issues. The largest issue is posed by the bandwidth, which for GPUs largely depends on the problem size. The more thread blocks that are in use, the higher the bandwidth. This is unlike CPUs where the problem can be relatively small and still utilize all ports. Therefore it is important to make measurements of the actual bandwidth for any given input size.

Since the computational intensity, that we computed earlier, is not particularly large for reasonable polynomial degrees it is unlikely we will be close to the maximal performance of the GPU P. We, therefore, expect the results to always be in the so-called memory-bound region where Roofline $= T \cdot B$ where B is the empirically measured bandwidth for each input size.

2.2.3 Reasons for Performance

It should also be mentioned why performance in this domain is important. To the bystander it might not seem like a large difference if a computation takes more or less time other than that you might have to wait a second or two longer. However, for large experiments and computations that take hours, or even days, a small increase in performance can save large amounts of research time. Additionally, high performance computers utilize a lot of power [3], so being able to make the computations faster also saves money and energy.

Chapter 3
Implementation

In this chapter, we present Nekbone more in detail as well as what programming languages and standards that are used. In particular we go though the optimizations performed on the matrix free Ax subroutine and the reasoning behind them. The code is available on Github at `https://github.com/MartinKarp/Nekbone/tree/cuda-openacc`.

3.1 Programming standards

The original Nekbone was entirely implemented in Fortran77 with a communication kernel in C and pure MPI for parallelization. From previous work by Gong et.al. most of the code has already been ported to GPU with OpenACC, however, this gave suboptimal performance compared to a mixed approach where the main computational kernel Ax was ported with CUDA Fortran [4]. GPUs can be programmed in several different ways, but in this report we will continue with utilizing CUDA and OpenACC as Gong did in their previous work.

3.1.1 OpenACC

OpenACC is a pragma based approach to GPU programming mainly developed by PGI and works similarly to OpenMP, but for GPUs. It is relatively simple to learn and does not require large changes in the original program. One simply adds pragmas around regions one wants to run on the GPU and then the compiler creates an autogenerated CUDA kernel that can be run on the GPU.

3.1.2 CUDA

CUDA gives the programmer more control and is developed by NVIDIA for their GPUs. It is currently the dominating GPU programming approach and very similar syntax to C.

In this approach the developer needs to write their own kernel in CUDA and then call it in their original program. This leads to more control over the kernel and execution on the GPU, but at the expense of ease of use and portability. There exists a CUDA Fortran version developed by PGI, and it supports writing CUDA kernels in a Fortran-like language. CUDA Fortran however, lacks certain functionality compared to CUDA C and certain pragmas such as `#pragma unroll` which specifies to the compiler to unroll a specific loop, enabling more compiler optimizations.

Algorithm 2 Naive Ax implementation.

```
procedure AxNaive
Input: $G \in R^{6\times n}, D \in R^{N\times N} \underline{u} \in R^n$
Output: $w \in R^n$
    Spawn a thread block for each element $e$ with $x$ number of threads and indices $i, j, k$.
    Calculate stride needed to go through element of size $N^3$ with $x$ threads.
    for each thread do
        while $\{i, j, k\} \in [1, N]^3$ do
            Calculate first derivatives of $\underline{u}(i, j, k, e)$ with $G$ and $D$.
            $i, j, k \leftarrow i, j, k$ + stride.
        end while
        Call syncThreads().
        Reset $i, j, k$.
        while $\{i, j, k\} \in [1, N]^3$ do
            Calculate second derivatives of $\underline{u}(i, j, k, e)$ with $D$.
            Store result in $w(i, j, k, e)$.
            $i, j, k \leftarrow i, j, k$ + stride.
        end while
    end for
    Perform the Gather-Scatter operation $Q$ and the restriction operation $R$.
end procedure
```

3.2 Optimizations

First, an initial implementation in CUDA was made and we then moved on to optimizing the Ax kernel, especially for locality, similar to the work by Świrydowicz et. al. [11], but on the original Nekbone code and not using OCCA, which they used. Some work on optimizing the kernel has already been done by Gong et. al. by utilizing shared memory to a larger extent. However, in this work we took the optimizations one step further and revised the thread structure as well as optimized the kernel for registers and shared memory.

3.2.1 Initial implementation

The first approach was simply to port the Ax subroutine from OpenACC to CUDA Fortran. There was already an implementation by Jing et al. available, but as to familiarize oneself with the code and subroutine, another one was made. This first approach is described with pseudocode in algorithm 2. In the algorithm, the notation is consistent with the Cost analysis section where the number of GLL points is N and the number of elements is E. The total number of points is then $n = N^3 \cdot E$. To illustrate the optimizations performed it is not necessary to exactly describe the matrix operations and those operations are therefore omitted from the pseudocode. As can be seen in the code, the different threads compute the element with some stride. This means that one thread can work on several seemingly random positions in the element, making data reuse hard.

3.2.2 Shared memory

The most obvious flaw with algorithm 2 is, therefore, the lack of data locality. In its current form, the data from global memory is accessed several times. By instead loading the data into shared memory in the beginning of the algorithm, performance gains can be made. In other words, you load $\underline{u}, D$ and w into shared memory and then do the exactly same algorithm with them instead. This is an optimization performed by Gong et al. However, this runs into some issues as well, mostly because storing all of these arrays in shared memory is not feasible. There is simply to much data when the polynomial degree increases. Our next optimization is therefore to avoid some of this by changing the structure of the thread block.

Algorithm 3 Ax subroutine with 2D-thread structure

```
procedure $Ax$2D
Input: $G \in R^{6\times n}, D \in R^{N\times N} \underline{u} \in R^n$
Output: $w \in R^n$
    Spawn a thread block for each element $e$ with $N \times N$ threads and indices $i, j$.
    Load $D$ into shared memory.
    Load $\underline{u}(i, j, :, e), w(i, j, :, e)$ into register arrays $u_r, w_r$.
    for each thread do
        Go layer by layer, $k \leftarrow 1$.
        while $k \leq N$ do
            Load $u_r(k)$ into shared memory.
            Call syncthreads().
            Calculate first derivatives of $u_r(k)$ with $G$ and $D$.
            Call syncthreads().
            Calculate second derivatives of $u_r(k)$ with $D$.
            Store results in $w_r(k)$.
            $k \leftarrow k + 1$.
        end while
        Load $w_r$ back into global memory.
    end for
    Perform the Gather-Scatter operation $Q$ and the restriction operation $R$.
end procedure
```

3.2.3 2D-thread structure and registers

Interesting from the paper by Świrydowicz et. al. [11] is especially the idea of using a 2D-thread structure for each element. This means that instead of allocating as many threads as possible per element and then computing each point in the elements with some stride, as is shown in Algorithm 2, each thread block instead uses a layer of $N \times N$ threads. This enables us to restructure the algorithm somewhat as can be seen in algorithm 3. Each thread then goes through its own specific pile of points and the thread block computes the element layer by layer in sync. This executes more `syncthreads()` operations that sync all the threads in a warp than before, but with the benefit that each thread executes more computations on its own column of nodal points. These points can be saved in a threads individual registers and higher performance can be achieved. Another thing worth noting is that the amount of data in shared memory is vastly decreased since we at any time only keep one layer of $\underline{u}$ in shared memory. The reason that we need some points in shared memory at all is that they are necessary to calculate the differentials of neighboring points.

Algorithm 4 Optimized CUDA C Ax subroutine.

```
procedure AxOpt
Input: const double* __restrict__ $G \in R^{6\times n}, D \in R^{N\times N}, \underline{u} \in R^n$
Output: double* __restrict__ $w \in R^n$
    Spawn a thread block for each element $e$ with $N \times N$ threads and indices $i, j$.
    Load $D$ into shared memory.
    Load $\underline{u}(i, j, :, e), w(i, j, :, e)$ into register arrays $u_r, w_r$.
    for each thread do
        Go layer by layer, $k \leftarrow 1$.
        #pragma unroll.
        while $k \leq N$ do
            Load $u_r(k)$ into shared memory.
            Call syncthreads().
            Calculate first derivatives of $u_r(k)$ with $G$ and $D$.
            Call syncthreads().
            Calculate second derivatives of $u_r(k)$ with $D$.
            Store results in $w_r(k)$.
            $k \leftarrow k + 1$.
        end while
        Load $w_r$ back into global memory.
    end for
    Perform the Gather-Scatter operation $Q$ and the restriction operation $R$.
end procedure
```

3.2.4 Unrolling and read-only arrays

The last part of the optimizations was for the CUDA C version where the code was compiled with `#pragma unroll` to specify loops where loop unrolling was permitted. This combined with specifying constant input arguments as read-only with the keywords `const __restrict__` enabled for more compiler side optimizations. The final pseudocode over the algorithm can be seen in algorithm 4. In these final optimizations some changes to the calculations were made as well, such as using `#pragma unroll` for vector/differential operations and changing the order of some operations. The main performance gains though were already made from the 2D thread structure and using shared memory and registers.

3.3 Communication kernel

The communication kernel it is not the focus of this report regarding optimization, however, we are very much interested in seeing how this affects the performance for multi GPU systems as we move forward. The communication kernel executes the gather-scatter operation which implicitly executes the multiplication of the gather-scatter matrix $\boldsymbol{Q}$. It is implemented in C and uses MPI for communication.

Chapter 4
Evaluation

In this chapter, we first present the measurements made to asses the performance of Nekbone as well as the experimental setup on the supercomputers Kebnekaise in Umeå and Piz Daint in Switzerland. Following this we present the results from the measurements and discuss the performance achieved. We also discuss possible future developments, bottlenecks, and comment on the measurements themselves.

4.1 Measurements

The measurements of the code were performed on Nekbone running the Conjugate Gradient solver for 100 iterations. Important for the roofline model was a proper analysis of the Bandwidth of the GPU, which was done by running a `cudaMemcpy` for each essential load and store of a variable and running this 100 times. With this the real utilized global bandwidth could be computed and with this the roofline for the different input sizes was also calculated.

Several different versions of the code were tested. First, the the original OpenACC version was tested to get the base case. The other version that were measured was the original CPU version and Gong's initial CUDA Fortran implementations as seen in Algorithm 2. After this, the initial shared memory optimization by Gong et. el. was measured. Lastly, our optimized versions of CUDA Fortran as well as CUDA C were measured. These implemented the 2D thread structure and other optimizations, pseudocode is shown in Algorithm 3 for the optimized CUDA Fortran and Algorithm 4 for CUDA C.

For the performance benchmarks, we compared the achieved performance for different number of elements in the range $2^8 - 2^{13}$ with a polynomial polynomial order of 9 and $N = 10$ GLL points. The reason for this was that measurements of different polynomial orders might be misleading and in Nek5000 polynomial orders above 10 is rarely used. The main reason for this is that the timesteps become to small because of various constraints on the dynamic system such as the CFL condition, which relates the size of the time step to the spatial resolution [1].

Measurements for multiple GPUs were also performed to evaluate the communication kernel and its impacts on the scaling. Both weak and strong scaling benchmarks were made and they were also compared to the peak performance for multiple nodes running in sync as well as the peak performance on a single node. These measurements were also made for 10 GLL points.

4.2 Experimental Setup

The tests were first and foremost prototyped on the cluster Kebnekaise's GPU nodes which are equipped with one Intel Xeon Gold 6132 with 28 cores @ 2.6GHz and 2 Nvidia v100 GPUs. The code was then compiled with the PGI compiler version 18.7 and CUDA version 9.2.

The multi GPU measurements were made on Piz Daint in Switzerland which, at the time of writing, is the 6th most powerful supercomputer in the world [2]. It is equipped with cray XC50 compute nodes, each node equipped with a 12 core Intel Xeon E5-2690 v3 @ 2.6GHz and one Nvidia Tesla P100 16GB GPU. On Piz Daint the version of the PGI compiler was 19.7 and the CUDA version was 10.1.

The theoretical peak performance for a V100 GPU (used at Kebnekaise) and a P100 GPU (Piz Daint) are up to 7 Tflops and 4.7 Tflops of double-precision performance respectively. In addition to this, the theoretical max bandwidth is 900 GB/s for a V100 and 732 GB/s for a P100. This is only a theoretical maximum though and we will base all benchmarks on empirical measurements.

4.3 Results

In this section we present the results from the single GPU measurements on both Piz Daint and Kebnekaise as well as the rooflines. The scaling benchmarks from Piz Daint are also presented.

4.3.1 Single GPU performance

First off, we performed single GPU benchmarks to evaluate the actual performance increase for each optimization on Kebnekaise. From this we obtained the plot shown in Figure 4.1. Similar measurements were made on Piz Daint and the result is shown in Figure 4.2. The versions plotted in Figure 4.1 is the original CPU version, OpenACC version as well as the shared memory optimization by Gong et.al. (CUF sh. mem.) and our optimized CUDA C (CUDAC) as well as our optimized CUDA Fortran version (CUF-opt). In Figure 4.2 the same versions are measured except for the CPU version and instead the original CUDA Fortran implementation by Gong was measured (CUF).

4.3.2 Roofline

The performance of our CUDAC version was compared to the measured roofline and is plotted for both a V100 GPU at Kebnekaise and a P100 at Piz Daint in Figure 4.3 and 4.4 respec-

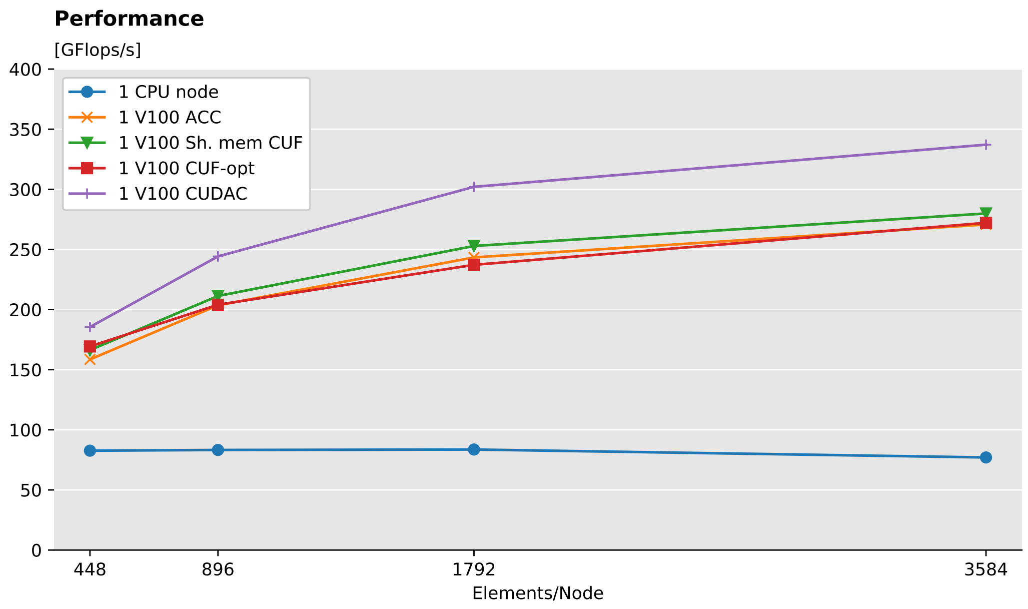


Figure 4.1: Performance of different versions on one V100 on Kebnekaise together with the performance of one CPU node run with MPI for parallelization and 28 cores.

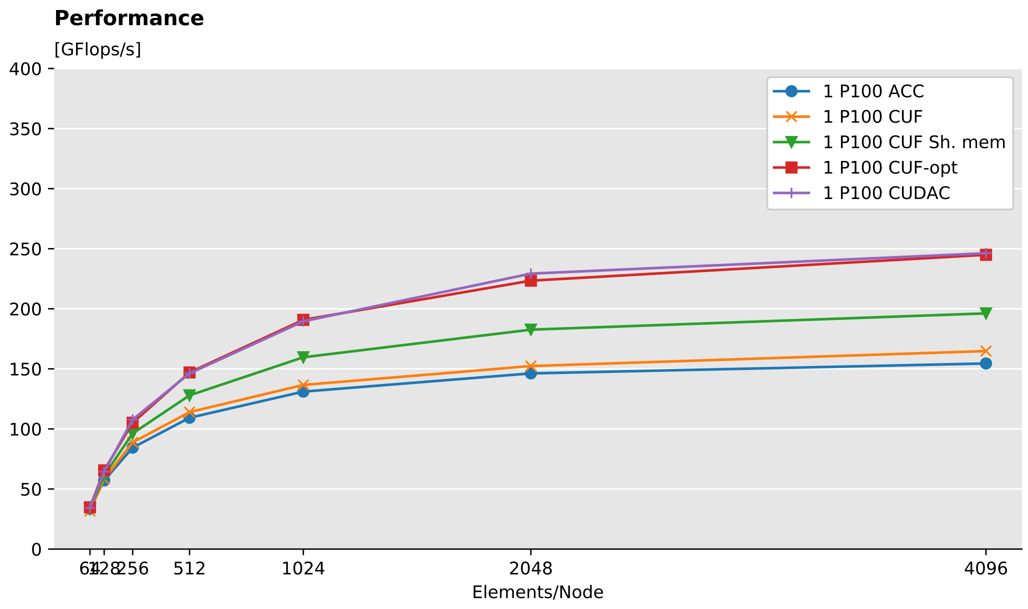


Figure 4.2: Performance of different versions on one P100 on Piz Daint.

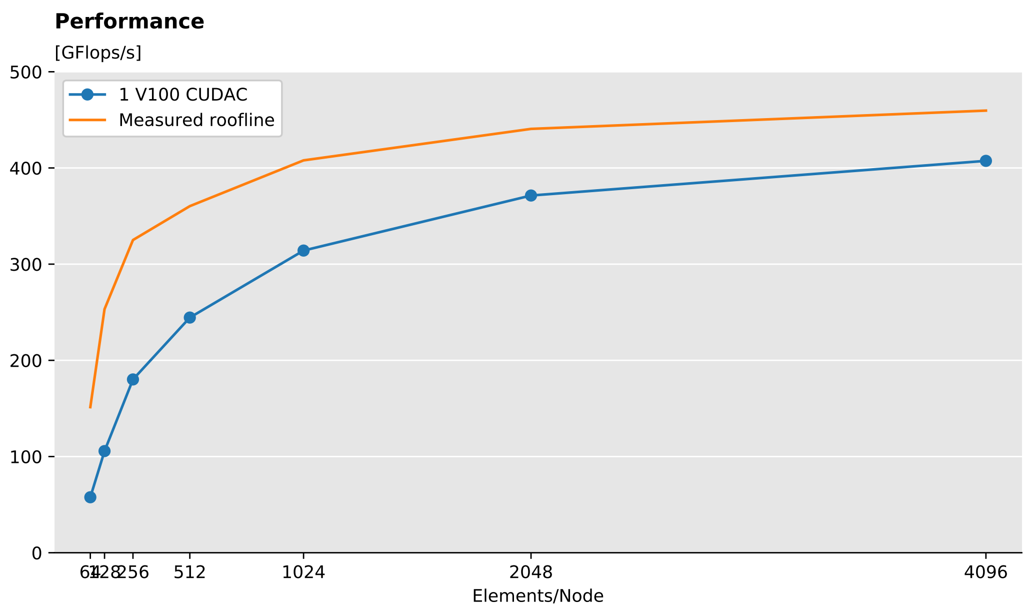


Figure 4.3: Roofline over single GPU performance for one V100 GPU on Kebnekaise.

tively. Note that the communication in this comparison was disregarded as the roofline is aimed to show how close to single GPU peak performance we are, not taking the communication into account.

4.3.3 Multi GPU Performance and Scaling

We also measured Multi GPU performance on Piz Daint. The performance of both strong and weak scaling is shown in Figure 4.5 and 4.6. Another plot, however, is the normalized performance increase per node compared to single GPU performance i.e. the parallel efficiency and is shown in Figure 4.7 and 4.8. To compare this to the very minimum amount of communication overhead needed between nodes, the same measurements were performed but without the gather-scatter operation. The parallel efficiency of those measurements can be seen in Figure 4.9 and 4.10.

4.4 Discussion

In this section we discuss the results and also bring up future developments in the area. We also compare the performance to previous work.

4.4.1 Single GPU performance and Roofline

First, if we look at the performance on Piz Daint and Kebnekaise in Figures 4.1 and 4.2 we see a clear difference between the different versions of the computing kernel. In both cases our CUDAC version gives the best performance while the OpenACC version gives the worst

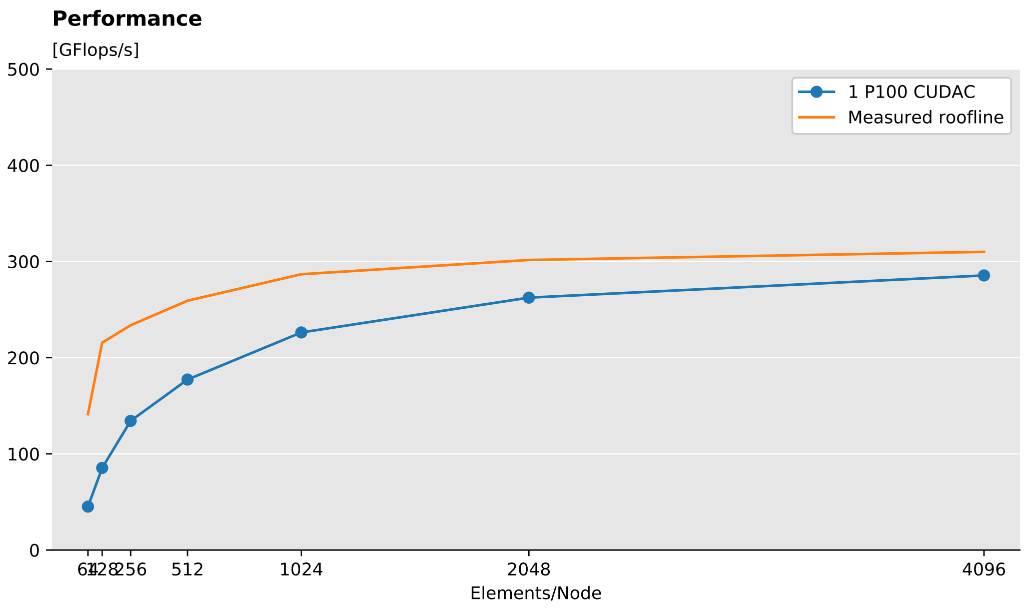


Figure 4.4: Roofline over single GPU performance for one P100 GPU on Piz Daint.

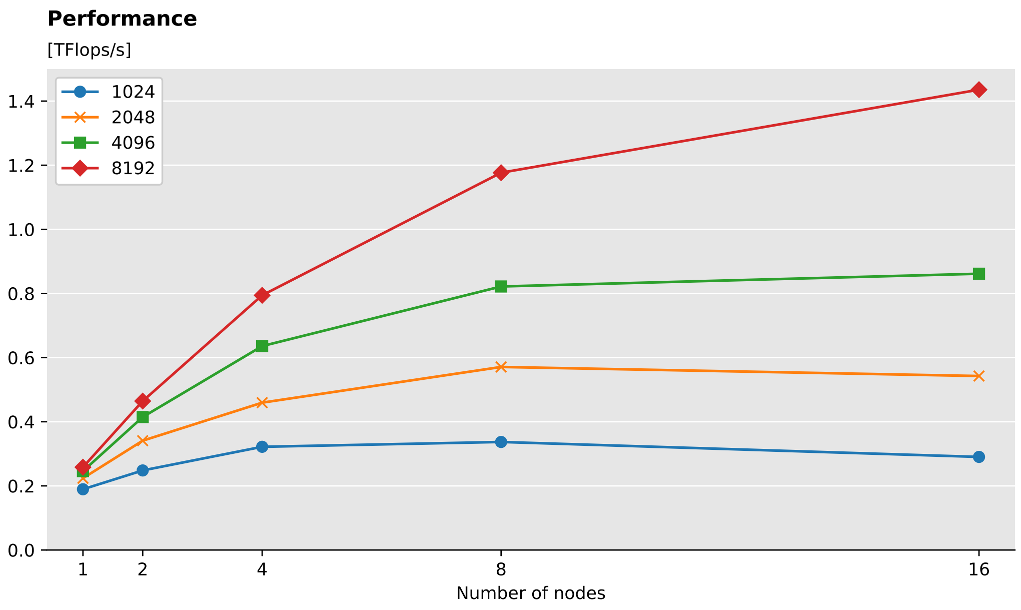


Figure 4.5: Strong scaling multi GPU performance on Piz Daint. The number of nodes is increased while the number of elements per node is constant.

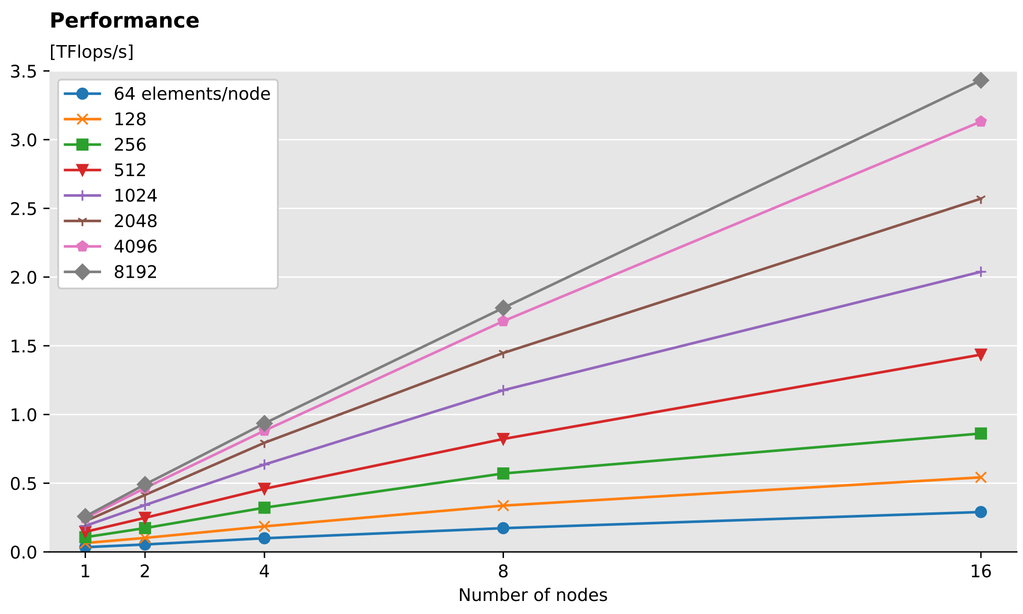


Figure 4.6: Weak scaling multi GPU performance on Piz Daint. The number of nodes is increased while the total number of elements is constant.

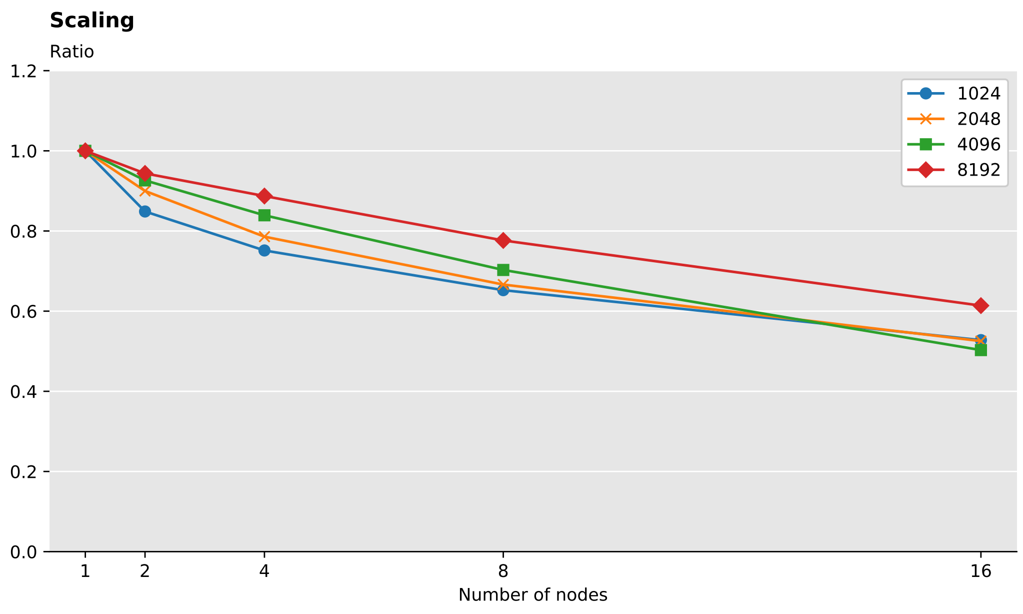


Figure 4.7: Strong scaling as a fraction of what is achieved on one node. This illustrates how the performance scaling is affected by increasing the number of nodes.

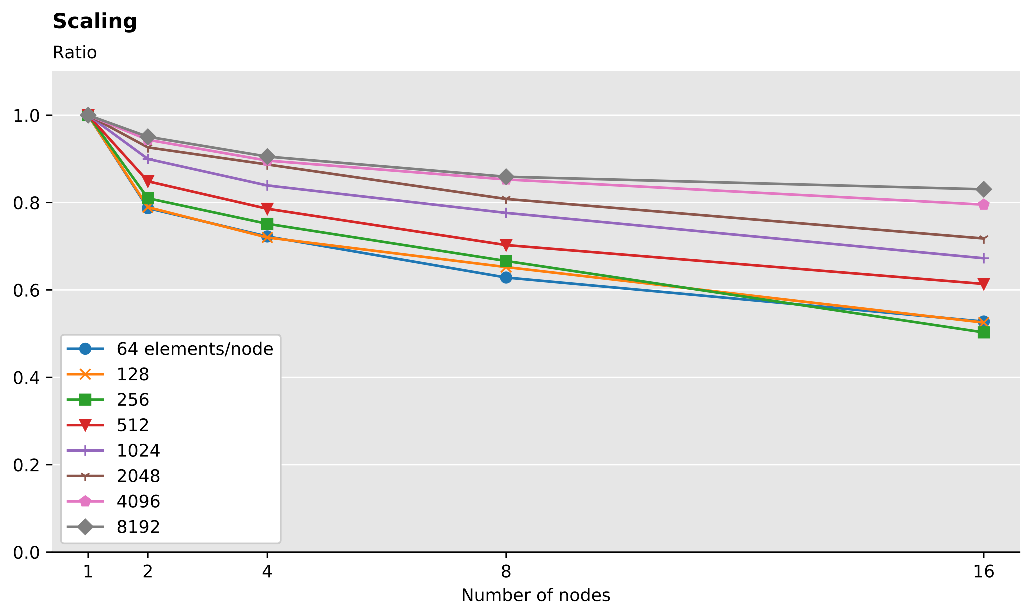


Figure 4.8: Weak scaling as a fraction of what is achieved on one node. This illustrates how the performance scaling is affected by increasing the number of nodes.

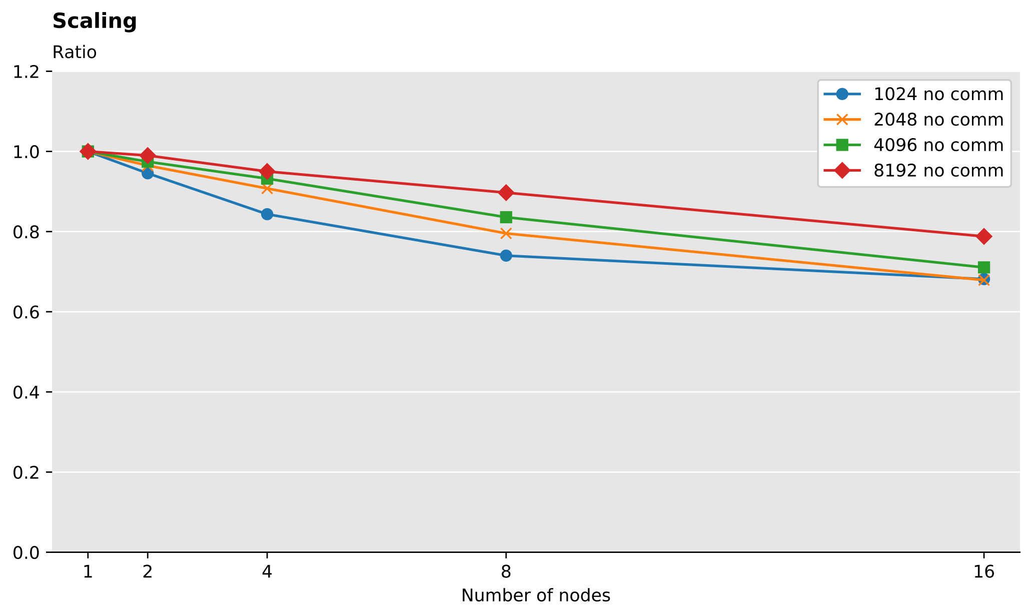


Figure 4.9: Scaling as a fraction of what is achieved on one node for the strong case, but without the gather-scatter operation. This illustrates how the performance scaling is affected by increasing the number of nodes.

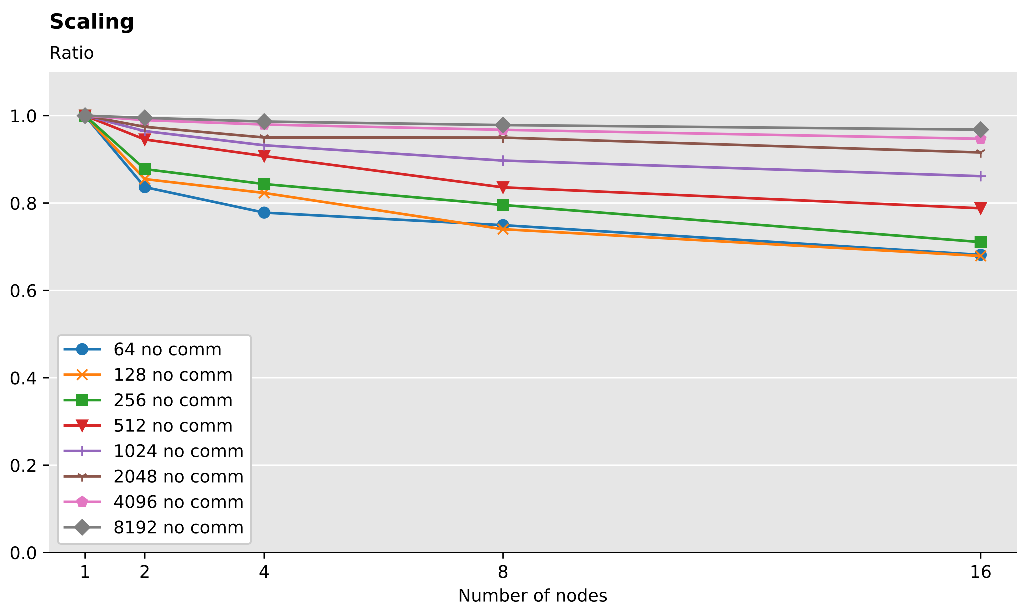


Figure 4.10: Scaling as a fraction of what is achieved on one node for the weak case, but without the gather-scatter operation. This illustrates how the performance scaling is affected by increasing the number of nodes.

results. This is not a surprise since OpenACC creates an auto-generated CUDA kernel that while our CUDAC kernel is optimized to use registers and shared memory as much as possible.

In particular the difference between using global and shared memory and registers is clearly illustrated in both figures. If we look at the performance of Gong's shared memory version it is evident that large performance gains are made when you utilize the Global Memory bandwidth as little as possible. The extra performance when using registers then comes as a result of having even less data movement than the shared memory version. In addition, adding unrolling and similar to the CUDAC version gives it a boost over the optimized CUDA Fortran, however, for Piz Daint the performance difference is almost negligible. This can likely be attributed to improvements in CUDA Fortran in newer versions of the PGI compiler.

To see how close we are to peak performance, we need to look at the roofline as shown in Figures 4.3 and 4.4. Here our optimized CUDA C kernel is compared to the theoretical peak and for larger inputs than 1024 the performance is in the range 70-90% of the empirical peak performance. This is aligned quite well with the results from Świrydowicz [11] where they, with the similar optimizations, achieved performance on the roofline for polynomial degree 10 and 4096 elements. Since we utilize simple OpenACC kernels for all other functions as well as for the masking of the boundary it is to be expected that we do not quite achieve 100% of the roofline. For the P100 and input size 4096 we do achieve 92% though. This is comparable and very close to the theoretical maximum and close to optimal.

With this performance it is, therefore, expected that new optimizations for a single GPU will not yield very much performance gains. One possible improvement that could be made would be to improve the OpenACC computations and especially the masking of the boundary which lacks good data locality. Additionally, one could look more closely at the Hybrid Schwarz Multigrid Preconditioner where similar improvements can be made [6].

We should also point out that the Roofline only considers the accesses to global memory. We may be limited by shared memory or cache bandwidth instead. It is also possible that for small input sizes and by operating on the same vectors repeatedly the performance can be even higher because of an increased number of cache hits. The size of the cache on both the V100 and P100 is limited to 4-6 MB though, so this effect would be most noticeable for inputs smaller than 1024 elements in our case.

4.4.2 Multi GPU performance and Scaling

The performance on a single GPU then, is close to optimal. We will now instead look at how the performance of our optimized CUDA C kernel is affected if we utilize more nodes. Let us once again start by looking at the raw performance in Figures 4.5 and 4.6. In these plots it is evident that having a high occupancy on the GPU is essential for high performance. It is especially obvious in the strong scaling case that adding more GPUs does not necessarily reduce the runtime if the number of elements per node is too low. This is to be expected though and can partially be explained by looking at the single node performance in Figure 4.2 for different input sizes.This is unavoidable and we are, therefore, more interested in the communication overhead that is inflicted by adding more GPUs.

If we look at Figures 4.7 and 4.8 where the performance is plotted as a fraction of one node performance it is evident that there is a overhead that increases with more nodes. The impact of this is heavily affected by the number of elements on each GPU, but the impact is major regardless. The question then is how much of this is unavoidable? There is an inherent latency that needs to be considered combined with the gather-scatter operation that involves communication between the nodes. If we look at Figure s4.9 and 4.10 the difference to the previous figures is immediately visible. In these figures the gather-scatter operation is not computed so the performance impact here is purely related with the MPI calls needed to perform the conjugate gradient routine in sync. This in and of itself also impacts the performance, it is therefore not feasible to expect a better parallel efficiency than this for the current structure of the code. However, it can also be concluded that the communication kernel imposes a severe degradation of performance as more nodes are added. The multi GPU perfomance is limited by the latency between nodes to some extent, but compared to the communication it has a negligible effect.

Another aspect of the scaling is that the current CPU version scales well for between 30-100 elements per core [8]. This would imply that the performance of a GPU must be comparable or better for input sizes as low as 1000 per node. It is very hard to make a proper performance comparison between GPU and CPU, but in future work this is something that must be considered.

For future work an interesting aspect of the code to improve the communication kernel. It is severely limited by the bandwidth and it imposes a significant drop in performance. Rewriting the communication kernel is a key aspect to make Nek5000 run efficiently on GPUs for large scale simulations. Additionally, the communication problem can be expected to be even worse when using a preconditioner, something that is always used in the production code.

4.4.3 Measurement comments

An interesting thing to note is once again that the PGI compiler versions are different between Kebnekaise and Piz Daint. This is a likely explanation of why our optimized CUDA Fortran version on Daint performs almost as good as the CUDAC version. It is likely that newer versions of CUDA and better compiler optimizations for CUDA Fortran have made a large impact on the performance of CUDA Fortran. This might have affected the other measurements as well, but since the comparison between different optimizations was always made on the same machine this should not affect the general analysis.

Another aspect is that we limited our measurements to 10 GLL points in each direction, i.e. polynomial degree 9. Measurements for higher or smaller degrees can also be of interest. However, it is also important to take into account that using a too high polynomial can inflate the results because we will get an artificial higher computational intensity. Degrees higher than 11 or 12 are rarely/never used in real applications because of the severe impact on the time step. The code in its current state is usually run with degrees 7-9, and that is why we have focused on 10 GLL points in this report.

Chapter 5

Conclusions

In summary the Spectral Element method without a preconditioner is memory bound, but for large enough input sizes and reasonable polynomial degree a close to optimal implementation on one GPU can be made. Going forwards the main obstacles are instead related to the scaling on multiple nodes. The current communication kernel is not well fitted and imposes heavy losses on the weak/hard scaling capabilities on the code.

5.1 Relating to other work

As mentioned we have made a highly performant single GPU version of Nekbone. Compared to Gong et. al. this is a large performance increase. Comparing our work to Swirydowicz et. al. we closely match their performance and with the added benefit that instead of using OCCA we used CUDA and the real Nekbone application. In their work they optimized the Ax subroutine, but did not measure or implement it in the overall application which we have done here. In particular, the multi-node measurements and scaling evaluation presented in this work is novel.

5.2 Research questions

In this section we answer the research questions that were presented in the beginning of the report.

What performance can be achieved and is currently achieved in Nekbone on single and multiple GPUs?

Performance close to the theoretical roofline can be achieved on one GPU through a optimized CUDA kernel for the matrix multiplication Ax and a simple OpenACC approach for most of the other operations. As for multi GPU performance, the GPUs are heavily affected by the number of elements on the GPU as well as a communication overhead. This in particular affects the weak scaling. Exact performance measurements are shown in section 4.3.

What performance limiting factors are there for Nekbone?

The main limiting factors for Nekbone on GPUs is currently the communication kernel which has a significant impact on scaling performance. Making the communication more efficient is the most important next step to a highly performing GPU code for Nek5000.

5.3 Future Work

The most notable improvement that can be made is then a revision of the communication kernel. Enabling better multi GPU scaling needs to be the focus of future work. Small improvements can also be made regarding the single GPU implementation of Nekbone, but this would mostly be limited to the masking operation and other OpenACC operations. In addition to this, making an extensive comparison of the performance of CPU versus GPU would be interesting. Lastly, bringing the improvements made in the report into Nek5000 would tremendously improve the performance compared to a pure OpenACC approach for GPU programming.

References

[1] R. Courant, K. Friedrichs, and H. Lewy. On the partial difference equations of mathematical physics. *IBM Journal of Research and Development*, 11(2):215–234, 1967.

[2] Swiss National Supercomputing Centre CSCS. "piz daint", one of the most powerful supercomputers in the world. `https://www.cscs.ch/fileadmin/user_upload/contents_publications/factsheets/piz_daint/FSPizDaint_Final_2018_EN.pdf`, Accessed 4-5-2020.

[3] Gilles Fourestey, Ben Cumming, Ladina Gilly, and Thomas C. Schulthess. First experiences with validating and using the cray power management database tool, 2014.

[4] Jing Gong, Stefano Markidis, Erwin Laure, Matthew Otten, Paul Fischer, and Misun Min. Nekbone performance on gpus with openacc and cuda fortran implementations. *The Journal of Supercomputing*, 72, 07 2016.

[5] Magnus R Hestenes, Eduard Stiefel, et al. Methods of conjugate gradients for solving linear systems. *Journal of research of the National Bureau of Standards*, 49(6):409–436, 1952.

[6] James Lottes and Paul Fischer. Hybrid multigrid/schwarz algorithms for the spectral element method. *J. Sci. Comput.*, 24:45–78, 07 2005.

[7] Kyle Niemeyer and Chih-Jen Sung. Recent progress and challenges in exploiting graphics processors in computational fluid dynamics. *The Journal of Supercomputing*, 67:528–564, 09 2013.

[8] Nicolas Offermans, Oana Marin, Michel Schanen, Jing Gong, Paul Fischer, Philipp Schlatter, Aleks Obabko, Adam Peplinski, Maxwell Hutchinson, and Elia Merzari. On the strong scaling of the spectral element solver nek5000 on petascale systems. In *Proceedings of the Exascale Applications and Software Conference 2016*, pages 1–10, 2016.

[9] GUnnar Sparr and Annika Sparr. *Kontinuerliga System.* Number 9789144013558. Studentlitteratur AB, 2 edition, 2000.

[10] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[11] Kasia Świrydowicz, Noel Chalmers, Ali Karakus, and Tim Warburton. Acceleration of tensor-product operations for high-order finite element methods. *The International Journal of High Performance Computing Applications*, 33(4):735–757, 2019.

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2020-05-12

EXAMENSARBETE Optimization and Profiling of the Spectral Element Method on Graphics Processing Units
STUDENT Martin Karp
HANDLEDARE Niclas Jansson(KTH), Jonas Skeppstedt(LTH)
EXAMINATOR Krzysztof Kuchcinski (LTH)

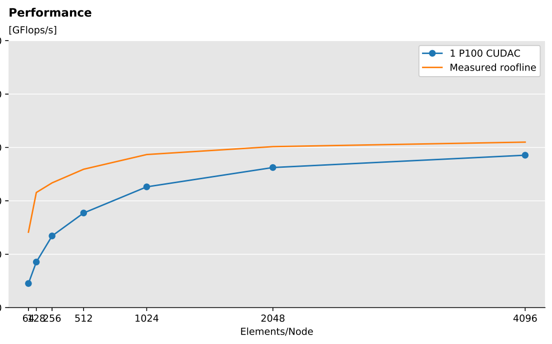
Snabba Vetenskapliga Beräkningar med Grafikkort

POPULÄRVETENSKAPLIG SAMMANFATTNING **Martin Karp**

Grafikkort, eller GPUer, används allt mer inom simulering och storskaliga beräkningar. Det här arbetet syftar till att undersöka hur man använder den här datorkraften effektivt för att kunna göra beräkningar mycket snabbare och mer effektiva än tidigare.

Många simuleringar kan i dagsläget ta flera timmar om inte dagar att genomföra. Även om man använder världens kraftfullaste dator finns det därför alltid ett behov av att optimera dessa beräkningarna. Tidigare har högre prestanda kunnat uppnås till viss del genom att själva datorerna blivit snabbare och bättre, men det är en trend som avtagit på senare år. Därför har intresset för att använda GPUer för vissa beräkningar ökat markant. De har tidigare använts främst till spel eller olika grafikberäkningar, men har på senare tid fått en större roll för andra beräkningar då de lämpar sig mycket bra till olika sorters matrisoperationer.

I det här arbetet utvärderas och optimeras en kod, Nekbone, som använt sig av vanliga processorer till att istället använda GPUer. Dock, tack vare ett blandat användande av GPU-programmerings metoderna CUDA och OpenACC behövdes inte enorma ändringar i koden göras, men mycket högre prestanda kunde uppnås ändå. Resultatet av denna insats blev en version av koden som på ett nästintill optimalt sätt utnyttjar datorkraften hos en GPU och vi visade också empiriskt att det stämde. För rimliga storlekar på problemen uppnådde vi över 80% av den empiriska högsta prestandan. Detta kan ses i bilden till höger.



För framtida arbeten med att optimera koden behöver man dock göra mycket arbete med att skala upp beräkningarna. Just nu är ett stort problem att även om koden fungerar mycket bra för en GPU så får man inte ut samma prestanda när man använder flera stycken.

Koden testades på den, i skrivande stund, sjätte starkaste superdatorn i världen, Piz Daint, och där såg man tydligt hur prestandaökningen man fick när man använder flera GPUer minskar drastiskt. I ett optimal scenario hade prestandan ökat linjärt med antalet GPUer, vilket vi inte observerade. I framtiden är det alltså aktuellt att förbättre kommunkationen mellan flera GPUer så att deras resurser kan utnyttjas på ett ännu effektivare sätt.