# A GPU-accelerated Jacobi preconditioner for high-order fluid simulations

Author: Jacob Wahlgren `<jacobwah@kth.se>`
Supervisors: Niclas Jansson & Martin Karp

January 11, 2022

**Abstract**

Next-generation scientific software must take advantage of acceleration to exploit emerging heterogeneous computing architectures. *Neko* is a new computational fluid dynamics code in development at KTH. The contributions of this work is to implement a GPU-accelerated Jacobi preconditioner in Neko to decrease the time to solve the pressure equation, and to compare the performance of different potential implementations on a NVIDIA A100 GPU. We find that a natural kernel design without use of shared memory performs the best. We conclude that on this device, automatic caching performs better than explicit use of shared memory, and that batching multiple computations in a single thread is not beneficial. Additionally, there is no apparent cache penalty for not matching execution blocks to geometric elements, allowing kernels to easily support arbitrary polynomial degrees.

## 1 Introduction

Emerging heterogeneous architectures offer large gains in computing power over conventional homogeneous architectures by the use of accelerators, but also pose a challenge as they require a paradigm shift in how computing systems are programmed. Fluid simulations are used in diverse fields such as mechanical engineering, medicine and climate science but applications are often limited by available computing capacity. Emerging heterogeneous computing systems, using e.g. graphical processing units (GPUs), may provide a solution by offering large computational capacity. However, codes must be adapted to use heterogeneous accelerators.

The HPC group at KTH is developing the next-generation computational fluid dynamics (CFD) solver Neko [2]. In the solver, the Poisson equation representing the pressure field is the most computationally demanding. Without a preconditioner it can take hundreds of iterations for the solver to converge. Therefore it is important to use an efficient preconditioner, which can reduce the number of iterations to the single digits. However, there is currently no support for preconditioners in Neko's GPU backend. The contributions in this work are:

- A Jacobi preconditioner for the CUDA backend in Neko.
- An empirical performance evaluation of different kernel designs on the NVIDIA A100 GPU.

## 2 Numerical method

The pressure $u$ is given by Poisson's equation.

$$\nabla^2 u = f \tag{1}$$

To solve the equation for a given space and time interval, an approximate numerical method is used.

### 2.1 Spectral element discretization

This section gives an overview of the spectral element method in the context of solving the Poisson equation in a CFD application. A comprehensive explanation of the method is available in [1, ch. 4].

We find an approximate solution $u^*$ by solving the following linear system. The Laplacian $\nabla^2$ is represented by the stiffness matrix $A$, and the solution $u^*$ is a finite dimensional vector.

$$Au^* = f \tag{2}$$

A piecewise polynomial approximation space of hexahedral elements is used. The polynomial degree $p$ results in $N = p + 1$ Gauss-Lobatto-Legendre quadrature points in each spatial dimension. Thus there are $EN^3$ degrees of freedom ($u^*$ is a vector of $EN^3$ components). In standard FEM, $N$ is usually in the range 2-4, while in a high-order/spectral method $N$ is usually around 8-12, achieving greater accuracy for a given mesh.

In FEM we can assemble $A$ into a sparse representation requiring $O(E(N^3)^2) = O(EN^6)$ memory. For higher-order polynomials that is not feasible. Instead we use a matrix-free form (or factored representation) like below in (3). The local stiffness matrix $A^e$ describes the geometry of element $e$, where each component represents one of the quadrature points. The tensor $G^e$ consists of diagonal matrices describing the mapping from $e$ to the reference element $\hat{e}$, the cube spanning $[-1, 1]^3$. The tensor $D$ consists of spatial derivative operators. Using this matrix-free formulation reduces the memory usage for $A$ to $O(EN^3)$.

$$A^e = D^T G^e D \tag{3}$$

Let $A_L$ be the diagonal block matrix of each local stiffness matrix $A^e$. In a parallel computation, each processor owns a subset of the elements and can compute a part of $A_L$ independently.

$$A_L = \text{diag}\{A^1, ..., A^E\} \tag{4}$$

Neighbouring elements share some of the quadrature points. To ensure the solution is continuous between different elements, we must ensure that the components representing the same point in space have the same value. Therefore a scatter-gather boolean matrix $Q$ is applied to $A_L$ to finally construct the global stiffness matrix $A$. This is the only step that requires communication between parallel processors.

$$A = Q^T A_L Q \tag{5}$$

Since $Q$ is large it is not explicitly formed either. In fact, the computations are always done in unassembled form. The solver only needs to compute matrix-vector products like $w = Ax$, which corresponds to the following operation in unassembled form.

$$w = Ax \iff w_L = QQ^T A_L x_L \tag{6}$$

## 2.2 Iterative method

To solve a large linear system such as (2), we use an iterative method. At a high level, an iterative method generates successively improving approximate solutions until the required precision (tolerance) has been reached. In Neko, a Krylov subspace method such as the conjugate gradient method is used. The error in the approximation is called the residual.

How fast an iterative method converges for some problem $Ax = b$ is dependent on the condition number of the problem. The condition number is a measure of how sensitive a system is to a small change in its output, and thus indicates how many iterations are required to reach a certain tolerance. To increase the performance of an iterative solver, a preconditioner can be applied to the system to reduce the condition number. Designing a good preconditioner is a trade-off between how long it takes to compute the preconditioner and how much it can reduce the condition number of the system.

Formally, the preconditioner $M$ of a matrix $A$ is a matrix such that the condition number of $M^{-1}A$ is smaller than the condition number of $A$. For efficient computation, finding the inverse $M^{-1}$ should be fast (in the extreme $M = A$ and we have to solve the system to solve the system...).

Perhaps the simplest variant is the Jacobi preconditioner $J = \mathrm{diag}(A)$. It is effective if the diagonal of $A$ is dominant, i.e. much larger than other parts of the matrix. Since $J$ is a diagonal matrix the inverse can be computed very efficiently.

$$(J^{-1})_{ii} = (J_{ii})^{-1} = (A_{ii})^{-1} \tag{7}$$

Let $J_L$ be the unassembled formulation of $J$, i.e. with duplicate values for shared points. We construct $J_L$ by assembling $\mathrm{diag}(A_L)$ and then applying the gather-scatter operation $QQ^T$.

$$J_L = QQ^T \mathrm{diag}(A_L) \tag{8}$$

The following closed form of $d^e = \mathrm{diag}(A^e)$ is used to compute the preconditioner $J_L$ [1, eq. (4.4.13)].

$$d^e_{ijk} = \sum_{l=0}^{N} \left[ \hat{D}^2_{li} (G^e_{11})_{ljk} + \hat{D}^2_{lj} (G^e_{22})_{ilk} + \hat{D}^2_{lk} (G^e_{33})_{ijl} \right] \tag{9}$$

## 3 Implementation

The implementation consists of two operations, to compute the inverse preconditioner $M^{-1}$ and to apply it to a vector. The basic structure for these is given in algorithm 1 and 2 respectively. The gather-scatter action $QQ^T$ is already implemented in Neko, as well as the basic component-wise invert and multiply operations. What remains is to find a suitable implementation of the diagonal assembly operation (9) in the form of a GPU-accelerated kernel.

```
Input: $G, D$
Output: Diagonal of $J^{-1}$ stored in $j$.
$j \leftarrow$ assemble diag($A_L$) using $G$ and $D$
apply $QQ^T$ to $j$
for $i \leftarrow 1$ to $EN^3$ do
    $j_i \leftarrow 1/j_i$
end
```

**Algorithm 1:** Construct Jacobi preconditioner.

```
Input: $j, r$
Output: $J^{-1}x$ stored in $z$.
for $i \leftarrow 1$ to $EN^3$ do
    $z_i \leftarrow j_i \cdot r_i$
end
```

**Algorithm 2:** Apply Jacobi preconditioner, i.e. solve $Jz = r$.

### 3.1 Diagonal assembly kernels

A major consideration in the design of a kernel is how to divide the work across computation threads and blocks. Each block consists of up to 1024 threads and has a fast shared memory and cache. Several variants are tried to see which yield the best performance.

Each kernel variant is described in text and then illustrated by a simplified CUDA-like code snippet. The snippets only show computation in the first dimension, since the others follow the same pattern. The variables $i, j, k$ are the quadrature point indices and $e$ the element index.

**Natural kernel** A natural division of the work is to assign one quadrature point to each thread, and use one block per element. The code for this kernel is straight-forward, with one $N$-iteration loop per spatial dimension.

```
int e = blockIdx.x;
int k = threadIdx.z;
int j = threadIdx.y;
int i = threadIdx.x;

for (int l = 0; l < N; l++) {
  double g = G11[l + N*j + N*N*k + N*N*N*e];
  double t = D1[i + N*l];
  d += g*t*t;
}
```

**Shared memory kernel** Since each component of $G_{11}$, $G_{22}$ and $G_{33}$ are used by multiple threads in each block, efficient use of the shared memory is important to achieve high performance. To ensure that the shared memory is fully utilized, each thread in this kernel loads its quadrature point from each $G_{ii}$ into the shared memory before the computations are carried out. The threads are synchronized after the loads so that all data is initialized before being read.

```
__shared__ double G11e[N][N][N];
G11e[i][j][k] = G11[i + N*j + N*N*k + N*N*N*e];
__syncthreads();
for (int l = 0; l < N; l++) {
  double g = G11e[l][j][k];
  double t = D1[i + N*l];
  d += g*t*t;
}
```

**Extended kernel** A downside of the natural kernel is that it can only support $N \leq 10$ since the number of threads per block cannot exceed 1024. To work around this limitation, we instead assign to each block 1024 consecutive quadrature points, not necessarily all from the same element. The kernel code is the same, except for how the indices are calculated.

```
int idx = threadIdx.x + blockIdx.x * blockDim.x;
int e = idx / (N*N*N);
int k = idx / (N*N) % N;
int j = idx / N % N;
int i = idx % N;
```

**Wide kernel** CPUs can achieve higher performance when batching operations together due to e.g. SIMD execution. We attempt the same technique in the

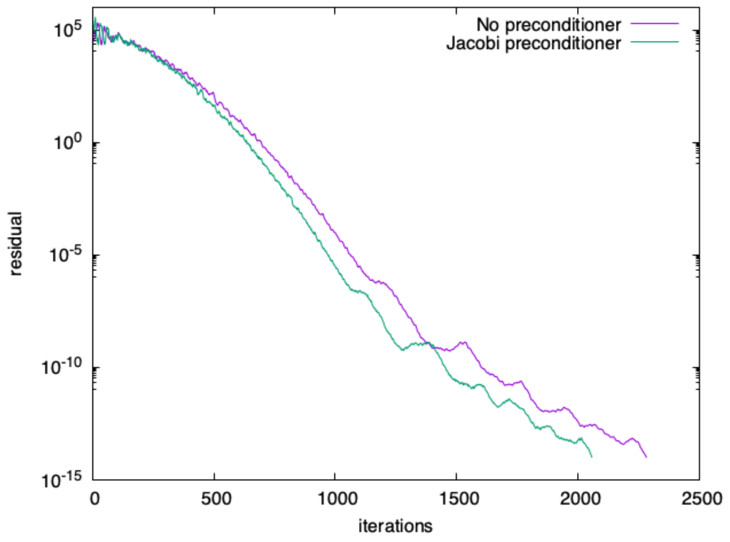

Figure 1: Solver residual with and without preconditioner.

GPU kernel by having each thread compute $N$ quadrature points instead of just one. Versions both with and without shared memory are implemented.

```
for (int l = 0; l < N; l++) {
  for (int i = 0; i < N; i++) {
    double g = G11[l + N*j + N*N*k + N*N*N*e];
    double t = D1[i + N*l];
    d[i] += g*t*t;
  }
}
```

## 4 Evaluation

The correctness and performance of the implementation is evaluated experimentally. The experiments are run on an Nvidia A100 40GB card with an AMD EPYC 7302P host.

### 4.1 Correctness

We evaluate the impact of using the Jacobi preconditioner on a sample case. The case is discretized with $N = 10$ and 32,768 elements, and we use a conjugate gradient iterative method. The residual size is saved in each iteration of the solver and shown in figure 1. It shows that the solver performs slightly better when using the Jacobi preconditioner. It requires 10% less iterations to reach a tolerance of $10^{-14}$.

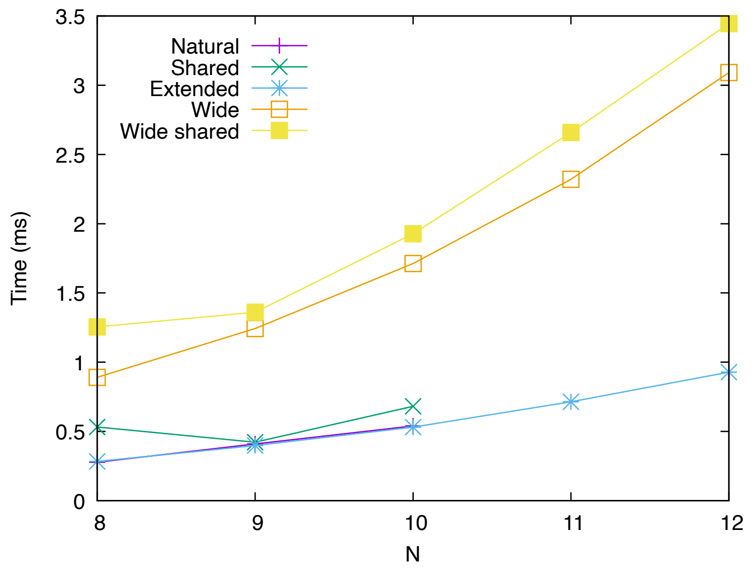

Figure 2: Average execution time of construct Jacobi operation with the different kernels variants, 4096 elements. See section 3.1 for a description of the kernels.

To ensure correctness we also compare the computed preconditioner $J^{-1}$ for each kernel to the previous non-accelerated Jacobi preconditioner implementation in Neko. We test all combinations of $N \in [8, 12]$ and $E \in 2^{[7,15]}$. The difference computed component-wise never exceeded $2 \cdot 10^{-13}$.

## 4.2 Performance

We evaluate the performance of the kernel versions by averaging the execution time of 100 warm cache runs for 4096 elements and different values of $N$. The results are shown in figure 2. The relative performance of the kernels were similar also for other problem sizes.

The best performing kernels are natural and extended which have indistinguishable results. The shared kernel performs a little worse, except for $N = 9$ where it matches the natural and extended. The two wide kernels perform much worse in all cases.

## 5 Conclusion

The natural and extended kernels were the best performing in all cases. There was no cache related penalty even though there was not exactly one block per element. Since the extended kernel can also support arbitrary $N$, it is used in the final implementation in Neko. Using shared memory explicitly lead to the same or worse performance in all cases, which means that the automatic cache of the A100 performs very well on this type of workload. Using wider kernels resulted in even worse performance, so there seems to be no gain from batching operations within a single thread.

## 5.1 Future work

While the Jacobi preconditioner slightly reduced the number of solver iterations, greater reductions are possible with a more advanced preconditioner such as a hybrid-Schwarz multigrid method [3]. An accelerated implementation of such a method is not yet a part of Neko.

The methods presented here only apply to hexahedral elements, though small modifications would make them work for deformed (i.e. curved) hexahedral elements as well.

## References

[1] M. O. Deville, P. F. Fischer, and E. H. Mund. *High-Order Methods for Incompressible Fluid Flow*. Cambridge Monographs on Applied and Computational Mathematics. Cambridge University Press, 2002. DOI: `10.1017/CBO9780511546792`.

[2] Niclas Jansson et al. *Neko: A Modern, Portable, and Scalable Framework for High-Fidelity Computational Fluid Dynamics*. 2021. arXiv: `2107.01243 [cs.MS]`.

[3] James W. Lottes and Paul F. Fischer. "Hybrid Multigrid/Schwarz Algorithms for the Spectral Element Method". In: *Journal of Scientific Computing* 24 (2005), pp. 45–78.