Second year report

Department of Earth Sciences
Institute of Geophysics
ETH Zurich

Fast and accurate Polynomial Transforms in fully spectral magnetohydrodynamics code QuiCC

Dmitrii Tolmachev

Prof. Dr. Andrew Jackson, ETH Zurich (supervisor)

Dr. Philippe Marti, ETH Zurich

6 July 2023

Contents

1 Introduction

Our research group is interested in the exploration of the process of generation, maintenance and evolution of magnetic fields in planetary and stellar cores. In the conventional regime of physics, thermal convection is believed to be the main driving force behind it. Convection can be described as a fluid movement that occurs due to the exertion of forces on the body, such as electromagnetic forces, buoyancy or gravity forces and forces related to the body rotation.

The mechanism of how convective motion is connected to the magnetic field is governed by the dynamo theory. The main idea behind it can be shortly explained as convection currents generated by the heat flow in the core of a rotating body forming spring-like patterns due to the Coriolis force, which acts as a circular electric current due to the fluid being conductive, generating the magnetic field. From the theoretical point of view, fluid movement is studied by the fluid dynamics branch of science with Navier-Stokes equations being one of the most important governing laws, heat transfer is governed by the transport equation and actual field formation obeys Maxwell laws and the induction equation.

To perform numerical simulations of the aforementioned problem, codebase named QuICC ("Quasi-Inverse Convection Code") has been developed by Dr. Philippe Marti. What separates QuICC from a conventional CFD is that it is based on a fully spectral method, that expands the unknown functions in terms of a set of orthogonal basis functions. The main advantage of this approach is that it has exponential error convergence for smooth functions which results in a much lower discretization requirements than finite-difference methods. Another advantage of QuICC is that it operates in full-sphere geometry, meaning that it is able to simulate full-body instead of just the shell. To do so, QuICC expands radial component on the basis of Jones-Worland polynomials and angular components on the basis of well-known spherical harmonics.

While conventional finite-difference and finite-element schemes have a long history of studies behind them and have plenty of fast implementations, spectral methods are way less developed. Of all spectral transforms, the Fourier transform is the most studied due to the simplicity of its basis. It is also the transform with one of the most efficient discrete implementations, called Fast Fourier Transform. Polynomial transforms, including Jones-Worland and Spherical Harmonics, most of the time have to use direct integration schemes through quadrature rules. In this document we present a polynomial connection approach to evaluation of polynomial transforms. Section 2 is dedicated to the analysis of the modern spectral transform algorithms and their implementations.

Modern High Performance Computing clusters switch to the GPUs as the source of their computational power as opposed to CPUs. GPUs are tailored for data-parallel algorithms where multiple cores perform the same operations on different memory locations. This allows simplifying the chip structure increasing the core count and decreasing power consumption and manufacturing price. However, making CPU code run within the GPU constraints is often a non-trivial task. Firstly, not all algorithms are easy to parallelize. Secondly, there is no single way to program GPU for different manufacturers - all of them try to promote their own solutions. To solve this issue a runtime optimization platform has been developed and improved in design since the research plan defence. With this platform it was possible to develop a new approach to the GPU kernel programming that works well but goes against what many people think is considered a good practice of GPU programming (including Nvidia engineers). Section 3 outlines the architectural design of this platform. Section 4 presents the range of algorithms rewritten in it, such as:

- VkFFT, an efficient GPU-accelerated multidimensional Fast Fourier Transform library.
- Polynomial connection algorithm code: tridiagonal and pentadiagonal banded parallel matrix solvers and multiplication algorithms, efficient block copying and scaling algorithms, Discrete Cosine Transforms code (part of VkFFT). The main focus of this PhD.
- Finite differences solver
- Non-uniform FFT calculation algorithm

In the section 5 a brief outline for the future work is given.

2 Quasi-Inverse Convection Code. Polynomial order connection algorithm

This section contains the numerical theoretical part of the research plan proposal, done in 2022. The system of Navier-Stokes equations has not been solved analytically except for some specific cases, so we will take a numerical approach to the problem. To do so, Quasi-Inverse Convection Code (QuICC) was designed by Dr. Philippe Marti[6]. QuICC has a modular structure for particular model definition, like Boussinesq approximation in full sphere geometry with toroidal-poloidal decomposition (spherical shell simulations are also possible). For spectral expansion of the state variables up to a spherical truncation L and radial truncation N, a scalar field can be written as:

$$f(r,\theta,\phi) = \sum_{l=0}^{L} \sum_{m=-l}^{l} \sum_{n=0}^{N} f_{l,n}^{m} G_n^l(r) Y_l^m(\theta,\phi) \tag{2.1}$$

here $G_n^l(r)$ - are polynomials used for radial expansion - for example, Jones-Worland polynomials[5], defined through the Jacobi polynomials as:

$$G_n^l(r) = W_n^l(r) = r^l P_n^{(-1/2,l-1/2)}(2r^2-1) \tag{2.2}$$

and $Y_l^m(\theta,\phi)$ - are unit normalized spherical harmonics, which can be defined through the Associated Legendre polynomials (which in turn can be connected to Ultraspherical polynomials, which in turn can be connected to Jacobi polynomials[8] for non-negative integer $l-m$):

$$Y_l^m(\theta,\phi) = (-1)^m \left(\frac{2l+1}{4\pi}\right)^{\frac{1}{2}} \left(\frac{(l-m)!}{(l+m)!}\right)^{\frac{1}{2}} P_l^m(cos\theta) e^{im\phi} = \tilde{P}_l^m(cos\theta) e^{im\phi} \tag{2.3}$$

$$P_l^m(cos\theta) = (-2)^m \frac{\Gamma(m+\frac{1}{2})}{\Gamma(\frac{1}{2})} (sin\theta)^m C_{l-m}^{m+\frac{1}{2}}(cos\theta) \tag{2.4}$$

$$C_{l-m}^{m+\frac{1}{2}}(x) = \frac{\Gamma(m+l+1)}{\Gamma(2m+1)} \frac{\Gamma(m+1)}{\Gamma(l+1)} P_{l-m}^{(m,m)}(cos\theta) \tag{2.5}$$

2.1 Jones-Worland polynomials

To accurately describe radial behavior of infinitely differentiable at all points (including center) function, it is important to take into the consideration the behavior of radial part ($r \in (0,1)$), which arises from infinite differentiability of the solution at the center:

$$f_l(r) \propto r^l \left(a_0 + a_1 r^2 + a_2 r^4 + \cdots + a_n r^{2n} + \cdots\right) \tag{2.6}$$

Jones-Worland polynomials[5] are designed to describe such functions and to exhibit orthogonality for the same l, following orthogonality definition of Jacobi polynomials:

$$\int_{-1}^{1} (1-x)^{-\frac{1}{2}} (1+x)^{l-\frac{1}{2}} P_i^{\left(-\frac{1}{2},l-\frac{1}{2}\right)}(x) P_j^{\left(-\frac{1}{2},l-\frac{1}{2}\right)}(x)\, dx = \frac{2^l}{(2i+l)} \frac{\Gamma\left(i+\frac{1}{2}\right)\Gamma\left(i+l+\frac{1}{2}\right)}{\Gamma(i+l)\Gamma(i+1)} \delta_{ij} \tag{2.7}$$

$$x = 2r^2 - 1 \tag{2.8}$$

$$\int_0^1 \left(2-2r^2\right)^{-\frac{1}{2}} \left(2r^2\right)^{l-\frac{1}{2}} P_i^{\left(-\frac{1}{2},l-\frac{1}{2}\right)}\left(2r^2-1\right) P_j^{\left(-\frac{1}{2},l-\frac{1}{2}\right)}\left(2r^2-1\right) 4rdr = \frac{2^l}{(2i+l)} \frac{\Gamma\left(i+\frac{1}{2}\right)\Gamma\left(i+l+\frac{1}{2}\right)}{\Gamma(i+l)\Gamma(i+1)}\delta_{ij} \quad (2.9)$$

$$\int_0^1 \left(1-r^2\right)^{-\frac{1}{2}} (r)^{2l} P_i^{\left(-\frac{1}{2},l-\frac{1}{2}\right)}\left(2r^2-1\right) P_j^{\left(-\frac{1}{2},l-\frac{1}{2}\right)}\left(2r^2-1\right) dr = \frac{1}{2(2i+l)} \frac{\Gamma\left(i+\frac{1}{2}\right)\Gamma\left(i+l+\frac{1}{2}\right)}{\Gamma(i+l)\Gamma(i+1)}\delta_{ij} \quad (2.10)$$

$$\int_0^1 \frac{1}{\sqrt{1-r^2}} W_i^l(r) W_j^l(r) dr = \frac{1}{2(2i+\alpha+\beta+1)} \frac{\Gamma(i+\alpha+1)\Gamma(i+\beta+1)}{\Gamma(i+\alpha+\beta+1)\Gamma(i+1)}\delta_{ij} \quad (2.11)$$

$$W_n^l(r) = P_n^{\left(-\frac{1}{2},l-\frac{1}{2}\right)}(2r^2-1) \quad (2.12)$$

where $\alpha = -\frac{1}{2}$ and $\beta = l - \frac{1}{2}$ - Jacobi polynomial $P^{(\alpha,\beta)}$order constants, used in 2.2 to define Jones-Worland polynomials, and $\frac{1}{\sqrt{1-r^2}}$ being the weight. Other combinations of (α, β) are also possible, this one is chosen as it will provide connection to FFT later. So, α is fixed and β varies similarly to l, which is tied to r. For $l = 0$, Jones-Worland polynomials are equal to the Chebyshev polynomials of first kind T (up to a scaling and change of variable):

$$W_n^0(r) = P_n^{\left(-\frac{1}{2},-\frac{1}{2}\right)}(2r^2-1) \quad (2.13)$$

$$T_n(x) = \frac{2^{2n}}{\begin{pmatrix} 2n \\ n \end{pmatrix}} P_n^{\left(-\frac{1}{2},-\frac{1}{2}\right)}(x) \quad (2.14)$$

The radial expansion of f - Jones-Worland Transform (JWT) is written as:

$$f_l^m(r) = \sum_n f_n W_n^l(r) \quad (2.15)$$

The projection integral to obtain the expansion coefficients in 2.15 is:

$$f_n = \int_0^1 \frac{1}{\sqrt{1-r^2}} f_l^m(r) W_n^l(r) dr \quad (2.16)$$

Below a plot of some Jones-Worland polynomials is given:

We will now cover two approaches to how the Jones-Worland polynomial expansion coefficients can be computed.

2.1.1 Quadrature rule for Jones-Worland polynomials

In this subsection we will describe the quadrature rule used to evaluate integrals on Chebyshev grid points, which are, in fact, zeros of Chebyshev polynomials $T_n(cos\theta) = cos(n\theta)$:

$$x_k = cos\left(\frac{2k-1}{2n}\pi\right) \quad (2.17)$$

for $x_k \in (-1, 1)$.

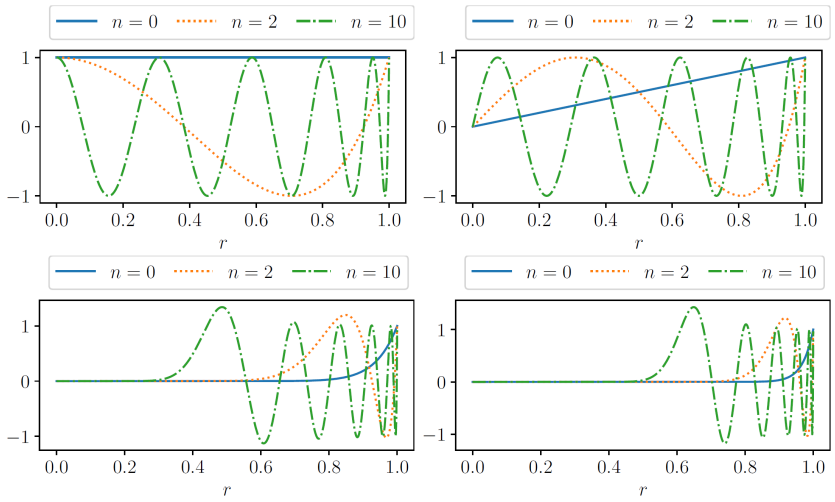

Figure 2.1: Jones-Worland polynomials for increasing $n = 0$, 2, 10 for (upper left) $l = 0$, (upper right) $l = 1$, (lower left) $l = 16$ and (lower right) $l = 32$. Image from [7]

While those grid points will not be optimal quadrature points for all degrees l (as they are roots only of W_n^0), it is unfeasible to have function values be measured on the different quadratures. So, we have to accept slower convergence and select a higher number of discretization points N_r. On this grid, projection integral 2.16 to obtain the expansion coefficients in 2.15 is written using Gauss quadrature rules:

$$f_n = \int_0^1 \frac{1}{\sqrt{1-r^2}} f(r) W_n^l(r)\, dr = \sum_{i=0}^{N_r} \omega_i f(r_i) W_n^l(r_i) \tag{2.18}$$

Here:

$$r_i = \sqrt{\frac{x_i + 1}{2}} \tag{2.19}$$

$$\omega_i = \frac{\pi}{n} \tag{2.20}$$

To calculate the values of Jones-Worland polynomials it is possible to use the following general recurrence relation for Jacobi polynomials:

$$a_n P_n^{(\alpha,\beta)}(x) = (b_n x + c_n) P_{n-1}^{(\alpha,\beta)}(x) + d_n P_{n-2}^{(\alpha,\beta)}(x) \tag{2.21}$$

$$\begin{aligned} a_n &= 2n(n+\alpha+\beta)(2n+\alpha+\beta-2) \\ b_n &= (2n+\alpha+\beta-1)(2n+\alpha+\beta)(2n+\alpha+\beta-2) \\ c_n &= (2n+\alpha+\beta-1)(\alpha^2-\beta^2) \\ d_n &= -2(n+\alpha-1)(n+\beta-1)(2n+\alpha+\beta) \end{aligned} \tag{2.22}$$

Full forward radial transform with quadrature rule is then computed as a matrix product:

$$\tilde{f}(r) = \boldsymbol{\mathcal{W}} \hat{f} \tag{2.23}$$

backward radial transform:

$$\hat{f} = \boldsymbol{\mathcal{W}}^T \boldsymbol{\omega} \tilde{f}(r) \tag{2.24}$$

here, $\tilde{f}(r)$ are the values of $f(r)$ on the grid, $\hat{f}$ the vector of spectral coefficients, $\boldsymbol{\omega}$ a diagonal matrix of the Chebyshev weights and column j of W contains $W_j(r_i)$

2.1.2 Polynomial order connection algorithm for Jones-Worland polynomials

A novel approach [7]to calculation of polynomial transforms by using the connection relations for the families of normalized Jacobi polynomials $\tilde{P}^{(\alpha,\beta)}$ instead of order:

$$\tilde{P}_n^{(\alpha,\beta)}(x) = \gamma_n^{(\alpha,\beta)} \tilde{P}_n^{(\alpha,\beta+1)}(x) + \zeta_n^{(\alpha,\beta)} \tilde{P}_{n-1}^{(\alpha,\beta+1)}(x) \tag{2.25}$$

$$\gamma_n^{(\alpha,\beta)} = \sqrt{\frac{2(n+\beta+1)(n+\alpha+\beta+1)}{(2n+\alpha+\beta+1)(2n+\alpha+\beta+2)}} \tag{2.26}$$

$$\zeta_n^{(\alpha,\beta)} = \sqrt{\frac{2n(n+\alpha)}{(2n+\alpha+\beta)(2n+\alpha+\beta+1)}} \tag{2.27}$$

and:

$$(1+x)\tilde{P}_n^{(\alpha,\beta)}(x) = \mu_n^{(\alpha,\beta)} \tilde{P}_n^{(\alpha,\beta-1)}(x) + \nu_n^{(\alpha,\beta)} \tilde{P}_{n+1}^{(\alpha,\beta-1)}(x) \tag{2.28}$$

$$\mu_n^{(\alpha,\beta)} = \sqrt{\frac{2(n+\beta)(n+\alpha+\beta)}{(2n+\alpha+\beta)(2n+\alpha+\beta+1)}} \tag{2.29}$$

$$\nu_n^{(\alpha,\beta)} = \sqrt{\frac{2(n+1)(n+\alpha+1)}{(2n+\alpha+\beta+1)(2n+\alpha+\beta+2)}} \tag{2.30}$$

It can be seen that:

$$\gamma_n^{(\alpha,\beta)} = \mu_n^{(\alpha,\beta+1)} \tag{2.31}$$

$$\zeta_n^{(\alpha,\beta)} = \nu_{n-1}^{(\alpha,\beta+1)} \tag{2.32}$$

The equation 2.25 is used to promote the β parameter to $\beta+1$. The equation 2.28 is used to promote the β parameter and get the required factor $r^l = \left(\frac{x+1}{2}\right)^{\frac{l}{2}}$ for even l (odd l case will be discussed later), present in Jones-Worland polynomials definition2.1. Both recurrence relations can be written in bidiagonal matrix form: 2.25 as an upper diagonal matrix $\mathbf{V}_l$ with the main diagonal consisting of $\gamma_n^{(\alpha,\beta-1)}$elements and the superdiagonal consisting of $\zeta_{n+1}^{(\alpha,\beta-1)}$elements for row n, 2.28 as a lower diagonal matrix $\mathbf{M}_l$ with the main diagonal consisting of $\mu_n^{(\alpha,\beta+1)}$elements and the subdiagonal consisting of $\nu_{n-1}^{(\alpha,\beta+1)}$elements for row n.

For Jones-Worland polynomials matrix $\mathbf{V}_l$ is used to connect $\sum_i c_i^{l-1} \tilde{P}_i^{(-\frac{1}{2},l-1-\frac{1}{2})}$ to $\sum_i d_i^l \tilde{P}_i^{(-\frac{1}{2},l-\frac{1}{2})}$ by a direct matrix multiplication:

$$\begin{pmatrix} \vdots \\ d_i^l \\ \vdots \\ \vdots \end{pmatrix} = \mathbf{V}_l \begin{pmatrix} \vdots \\ c_i^{l-1} \\ c_{i+1}^{l-1} \\ \vdots \end{pmatrix} \tag{2.33}$$

For Jones-Worland polynomials matrix $\mathbf{M}_l$ is used to connect $\sum_i c_i^{l-1} \tilde{P}_i^{(-\frac{1}{2},l-1-\frac{1}{2})}$ to $\sum_i d_i^l (1+x) \tilde{P}_i^{(-\frac{1}{2},l-\frac{1}{2})}$ by a direct matrix multiplication:

$$\begin{pmatrix} \vdots \\ d_i^l \\ \vdots \\ \vdots \end{pmatrix} = \mathbf{M}_l^{-1} \begin{pmatrix} \vdots \\ c_{i-1}^{l-1} \\ c_i^{l-1} \\ \vdots \end{pmatrix} \tag{2.34}$$

From 2.31 and 2.32 it is seen that:

$$\mathbf{M}_l = \mathbf{V}_l^T \tag{2.35}$$

By combining a sequence of $\mathbf{M}_l$ and $\mathbf{V}_l$, it is possible to obtain a Worland polynomial with an arbitrary harmonic degree l from $\tilde{P}_i^{(-\frac{1}{2},-\frac{1}{2})}$, for example a truncated at N_T Chebyshev expansion coefficients $c_i^{l=0}$:

$$f(x) \approx \sum_{i=0}^{N_T} c_i^{l=0} \tilde{P}_i^{(-\frac{1}{2},-\frac{1}{2})}(x) \tag{2.36}$$

can be propagated to obtain degree $\frac{l}{2}$ by matrix multiplying them with a set of $\mathbf{V}_{0\cdots\frac{l}{2}}$:

$$\mathbf{c}^{\frac{l}{2}} = \mathbf{V}_{\frac{l}{2}} \cdots \mathbf{V}_0 \mathbf{c}^0 \tag{2.37}$$

followed by backsolving sequence of $\mathbf{V}_l^T$:

$$\mathbf{V}_{\frac{l}{2}+1}^T \cdots \mathbf{V}_l^T \mathbf{c}^l = \mathbf{c}^{\frac{l}{2}} \tag{2.38}$$

Each multiplication by an upper diagonal matrix $\mathbf{V}_l$ corrupts the last matrix entry (as we don't have the value of $N_T + 1$ coefficient), so to have a Jones-Worland expansion of order N, N_T has to be at least $N + \frac{l}{2}$.

In order to improve the accuracy, we interleave the order in which $\mathbf{M}_l$ and $\mathbf{V}_l$ are applied:

$$\mathbf{c}^l = \left(\mathbf{M}_l^{-1} \mathbf{V}_{l-1}\right) \cdots \left(\mathbf{M}_1^{-1} \mathbf{V}_0\right) \mathbf{c}^0 \tag{2.39}$$

The final ingredients of the transform are the evaluation of 2.36 and normalization. By construction, 2.36 is the definition of Discrete Cosine Transform of type II and thus can be evaluated by the means of Fast Fourier Transform. This was the second main consideration (after the r^lbehavior) on how the Jones-Worland polynomials are constructed.

The backward transform is obtained by reversing the operations - now we multiply by $\mathbf{M}_l$ and backsolve for $\mathbf{V}_l$.

For odd l a special treatment is required as Jones Worland definition while being a polynomial in r will not be a polynomial in x if connected directly to Chebyshev polynomial of first kind T_n. By noting that:

$$W_n^1(r) = rP_n^{(-\frac{1}{2},\frac{1}{2})}\left(2r^2-1\right) = \frac{1}{2}(1+x)^{\frac{1}{2}} P_n^{(-\frac{1}{2},\frac{1}{2})}(x) \tag{2.40}$$

and:

$$\begin{aligned} &\int_0^1 \frac{1}{\sqrt{1-r^2}} W_i^1(r) W_j^1(r)\, dr = \\ &\frac{1}{4}\int_{-1}^1 \left(\frac{1+x}{1-x}\right)^{\frac{1}{2}} P_n^{(-\frac{1}{2},\frac{1}{2})} P_n^{(-\frac{1}{2},\frac{1}{2})} dx = \\ &\frac{1}{4}\int_{-1}^1 \left(\frac{1-x}{1+x}\right)^{\frac{1}{2}} P_n^{(\frac{1}{2},-\frac{1}{2})} P_n^{(\frac{1}{2},-\frac{1}{2})} dx \end{aligned} \tag{2.41}$$

Which is the weight function for Chebyshev polynomials of 4th kind W_n. So by reducing order not to W_n^0 (as in even case) but to W_n^1, we can still use Discrete Cosine Transform to evaluate Chebyshev expansion coefficients, but in this case, Discrete Cosine Transform will be of type IV.

2.2 Associated Legendre polynomials

The spherical harmonic expansion following 2.1 and 2.15 can be written as:

$$f(r,\theta,\phi) = \sum_{l=0}^{L}\sum_{m=-l}^{l} f_l^m(r) Y_l^m(\theta,\phi) \tag{2.42}$$

This is the definition of Spherical Harmonics Transform. $Y_l^m(\theta,\phi)$ are spherical harmonics, they form a complete set of orthogonal functions and are a good set to describe functions defined on the surface of a sphere. If we replace $\theta \to cos\theta$, we will get the definition $Y_l^m(\theta,\phi)$ based on Associated Legendre polynomials 2.5. Below we see $f_l^m(r)$ as a constant for all (θ,ϕ). The task is now to evaluate the inner part of SHT consisting of Associated Legendre polynomials, as the outer part is a regular Fourier Transform[9]:

$$f(\theta,\phi) = \sum_{m=-L}^{L}\left(\sum_{l=|m|}^{L} f_l^m \tilde{P}_l^m(cos\theta)\right) e^{im\phi} \tag{2.43}$$

The problem of finding coefficients f_l^m is then formulated as the following projection integral:

$$f_l^m = \int_0^{2\pi}\int_0^{\pi} f(\theta,\phi)\tilde{P}_l^m(cos\theta) e^{im\phi} sin\theta d\theta d\phi \tag{2.44}$$

Or if we separate:

$$f_l^m = \int_0^{2\pi}\left(\int_0^{\pi} f(\theta,\phi) e^{im\phi} d\phi\right)\tilde{P}_l^m(cos\theta) sin\theta d\theta \tag{2.45}$$

Let's denote:

$$\int_0^{\pi} f(\theta,\phi)e^{im\phi}d\phi = f^m(\theta) \tag{2.46}$$

Then Associated Legendre Transform (ALT) is then defined as:

$$f_l^m = \int_0^{2\pi} f^m(\theta)\,\tilde{P}_l^m(cos\theta)sin\theta d\theta \tag{2.47}$$

Both ALT and FT have weights equal to 1.

We can compare the Associated Legendre polynomials and Jones-Worland polynomials connection to Jacobi polynomials - Associated Legendre polynomials have $\alpha = \beta = m \in \mathbb{Z}$ and Worland polynomials have $\alpha = -\frac{1}{2}, \beta = l - \frac{1}{2}$.

We will now cover two approaches to how the Jones-Worland polynomial expansion coefficients can be computed.

2.2.1 Quadrature rule for Associated Legendre polynomials

The natural quadrature grid for Associated Legendre polynomials is Legendre grid (θ_i^{leg} are zeros of the Legendre polynomials, or P_l^0). On it the integral 2.47 Gauss quadrature is evaluated as:

$$f_l^m = \int_0^{2\pi} f^m(\theta)\,\tilde{P}_l^m(cos\theta)sin\theta d\theta = \sum_{i=0}^{N_\theta} \omega_i^{leg} f\left(\theta_i^{leg}\right)\tilde{P}_l^m\left(cos\theta_i^{leg}\right) \tag{2.48}$$

$$\omega_i^{leg} = \int_0^{2\pi} \frac{P_l(x)}{(x-x_i)P_l'(x)}dx \tag{2.49}$$

The expansion will be exact for polynomials of degree up to N_θ for N_θ nodes[9].

It is also possible to evaluate 2.47 on the Chebyshev grid using Clenshaw–Curtis quadrature:

$$f_l^m = \int_0^{2\pi} f^m(\theta)\,\tilde{P}_l^m(cos\theta)sin\theta d\theta = \sum_{i=0}^{2N_\theta} \omega_i^{CC} f\left(\theta_i^{CC}\right)\tilde{P}_l^m\left(cos\theta_i^{CC}\right)$$

with grid points and weights being:

$$\theta_i^{CC} = \frac{i\pi}{2N_\theta} \tag{2.50}$$

$$\omega_i^{CC} = \frac{2\varepsilon_i^{2N_\theta}}{N_\theta}\sum_{t=0}^{N_\theta} \varepsilon_t^{N_\theta}\frac{1}{1-4t^2}cos\left(\frac{it\pi}{N_\theta}\right) \tag{2.51}$$

with:

$$\varepsilon_i^I = \begin{cases} \frac{1}{2} & i = 0, i = I \\ 1 & 0 < i < I \end{cases} \tag{2.52}$$

From [4] can be seen that it is exact for polynomials of degree up to N_θ for $2N_\theta$ nodes, which means that it requires twice as many nodes as the Gauss-Legendre quadrature. The main advantage of the Clenshaw–Curtis quadrature

is that it has a connection to the Chebyshev grid, so 2.51 can be computed using very efficient Discrete Cosine Transforms[14].

To calculate values of high degree l, we use the same recursion relation 2.22 to connect Associated Legendre polynomials to the polynomials known analytically, for example:

$$P_m^m = \sqrt{\frac{1}{4\pi}\prod_{k=1}^{|m|}\frac{2k+1}{2k}}\left(1-x^2\right)^{\frac{|m|}{2}} \tag{2.53}$$

2.2.2 Polynomial order connection algorithm for Associated Legendre polynomials

It is possible to derive an order connection algorithm for Associated Legendre polynomials similar to 2.1.2[10]. The main difference is that now we have different values for α, β in the Jacobi polynomial definition. From 2.4 and 2.5 we can see that:

$$P_l^m(cos\theta) \underset{\theta}{\sim} (sin\theta)^m C_{l-m}^{m+\frac{1}{2}}(cos\theta) \underset{\theta}{\sim} (sin\theta)^m P_{l-m}^{(m,m)} \tag{2.54}$$

We will use the following Ultraspherical polynomial recurrence relations:

$$C_n^{m+\frac{1}{2}}(cos\theta) = \gamma_n^{m+\frac{1}{2}} C_n^{m+\frac{3}{2}}(cos\theta) + \zeta_n^{m+\frac{1}{2}} C_{n-2}^{m+\frac{3}{2}}(cos\theta) \tag{2.55}$$

$$\gamma_n^{m+\frac{1}{2}} = \frac{2m+1}{2n+2m+1} \tag{2.56}$$

$$\zeta_n^{m+\frac{1}{2}} = -\frac{2m+1}{2n+2m+1} \tag{2.57}$$

$$(sin\theta)^2 C_n^{m+\frac{1}{2}}(cos\theta) = \mu_n^{m+\frac{1}{2}} C_n^{m-\frac{1}{2}}(cos\theta) + \nu_n^{m+\frac{1}{2}} C_{n+2}^{m-\frac{1}{2}}(cos\theta) \tag{2.58}$$

$$\mu_n^{m+\frac{1}{2}} = \frac{(n+2m-1)(n+2m)}{(2m-1)(2n+2m+1)} \tag{2.59}$$

$$\nu_n^{m+\frac{1}{2}} = -\frac{(n+1)(n+2)}{(2m-1)(2n+2m+1)} \tag{2.60}$$

These two equations have a form resembling 2.25 and 2.28. Both recurrence relations can be written in matrix form: 2.55 as an upper triangular matrix $\mathbf{V}_m$ with the main diagonal consisting of $\gamma_n^{m-\frac{1}{2}}$elements and the second superdiagonal consisting of $\zeta_{n+2}^{m-\frac{1}{2}}$elements for row n, 2.58 as a lower triangular matrix $\mathbf{M}_{m+2}$ with the main diagonal consisting of $\mu_n^{m+\frac{3}{2}}$elements and the second subdiagonal consisting of $\nu_{n-2}^{m+\frac{3}{2}}$elements for row n. A subsequent application of $\left(\mathbf{M}_{m+2}^{-1}\mathbf{V}_m\right)$ performs a transformation from $C_n^{m+\frac{1}{2}}(cos\theta)$ to $(sin\theta)^2 C_n^{m+\frac{5}{2}}(cos\theta)$ or, after necessary scaling according to 2.4, $P_{n+m}^m(cos\theta)$ to $P_{n+m}^{m+2}(cos\theta)$. Backward transform is obtained by inverting the operations. It is also possible to accelerate the connection matrices calculation at a cost of accuracy[15].

As we see, in case of Associated Legendre polynomials connection is also performed with a step of 2, so it will reduce polynomials with even order m to $P_l^0(cos\theta)$ and polynomials with odd order m to $P_l^1(cos\theta)$. The resulting expansions can be written as:

$$f(x) = \sum_{i=0}^{N_T} c_i^0 \tilde{P}_i^0(x) \tag{2.61}$$

$$f(x) = \sum_{i=0}^{N_T} c_i^1 \tilde{P}_i^1(x) \tag{2.62}$$

To determine the correct N_T, similarly to 2.1.2, we note that each multiplication corrupts the last two entries, so N_T has to be at least $N + \frac{m}{2} + 1$.

The question that remains is how to evaluate 2.61 and 2.62, as they do not have a direct connection to a DCT or FT. We have two options here:

- Use quadrature rule 2.2.1 for $m = 0$.
- Use Chebyshev–Legendre transform that expresses $P_i^0(cos(\theta))$ as $T_i(cos(\theta))$ and $P_i^1(x)$ as $sin\theta U_i(cos(\theta))$. Here T_i and U_i are Chebyshev polynomials of first and second kind.

Let's discuss Chebyshev–Legendre transform in more detail.

2.2.3 Chebyshev–Legendre transform

The main advantage of this approach is, as was stated before, that on the Chebyshev grid we can perform efficient and accurate Chebyshev polynomial expansions by the means of DCT. Luckily, there exist an analytic expression for the Chebyshev–Legendre connection operator (we will demonstrate it for $P_i^0(cos(\theta))$ and $T_i(cos(\theta))$).[1, 8]

$$P^0 = TR_P^T \tag{2.63}$$

$$T = P^0 R_T^P \tag{2.64}$$

$$\pi R_P^T = \begin{pmatrix} \Lambda(0)^2 & 0 & \Lambda(1)^2 & 0 & \Lambda(2)^2 & \ddots \\ & 2\Lambda(0)\Lambda(1) & 0 & 2\Lambda(1)\Lambda(2) & 0 & \ddots \\ & & 2\Lambda(0)\Lambda(2) & 0 & 2\Lambda(1)\Lambda(3) & \ddots \\ & & & \ddots & \ddots & \ddots \end{pmatrix} \tag{2.65}$$

$$R_T^P = \begin{pmatrix} 1 & 0 & -\frac{5\Lambda(0)\Lambda(\frac{1}{2})}{6} & 0 & -\frac{\Lambda(1)\Lambda(\frac{3}{2})}{10} & \ddots \\ & \frac{\sqrt{\pi}}{\Lambda(1)} & 0 & -\frac{9\Lambda(0)\Lambda(\frac{3}{2})}{20} & 0 & \ddots \\ & & \frac{\sqrt{\pi}}{\Lambda(2)} & 0 & -\frac{5\Lambda(0)\Lambda(\frac{5}{2})}{7} & \ddots \\ & & & \ddots & \ddots & \ddots \end{pmatrix} \tag{2.66}$$

Here $\Lambda(k) = \frac{\Gamma(k+\frac{1}{2})}{\Gamma(k+1)}$. Indices of these matrices can be written as ($0 < i < j$ and $i + j$ even):

$$\left(R_P^T\right)_{ij} = \frac{2}{\pi}\Lambda\left(\frac{j-i}{2}\right)\Lambda\left(\frac{j+i}{2}\right) \tag{2.67}$$

$$\left(R_T^P\right)_{ij} = \frac{-j(i+\frac{1}{2})}{(j+i+1)(j-i)}\Lambda\left(\frac{j-i-2}{2}\right)\Lambda\left(\frac{j+i-1}{2}\right) \tag{2.68}$$

So combining order connection and evaluating 2.63 with 2.61 and DCT-II we can obtain the Associated Legendre transform expansion coefficients on the Chebyshev grid.

2.2.4 The Toeplitz–Hankel approach to Chebyshev–Legendre transform

The paper of Townsend [13] proposes a simple and accurate method of evaluating R_P^T connection. They state that R_P^T can be expressed as the upper-triangular component of a diagonally scaled Hadamard (entry-wise) product of a Toeplitz matrix and a Hankel matrix:

$$R_T^P = D\left[T \circ H\right] \tag{2.69}$$

$$T = \begin{pmatrix} \Lambda(0) & 0 & \Lambda(1) & 0 & \Lambda(2) & \ddots \\ & \Lambda(0) & 0 & \Lambda(1) & 0 & \ddots \\ & & \Lambda(0) & 0 & \Lambda(1) & \ddots \\ & & & \ddots & \ddots & \ddots \end{pmatrix} \tag{2.70}$$

$$H = \begin{pmatrix} \Lambda(0) & \Lambda(\frac{1}{2}) & \Lambda(1) & \dots \\ \Lambda(\frac{1}{2}) & \Lambda(1) & \Lambda(\frac{3}{2}) & \iddots \\ \Lambda(1) & \Lambda(\frac{3}{2}) & \Lambda(2) & \iddots \\ \vdots & \iddots & \iddots & \iddots \end{pmatrix} \tag{2.71}$$

$$D = diag(\frac{1}{\pi}, \frac{2}{\pi}, \frac{2}{\pi}, \dots) \tag{2.72}$$

here $\circ$ is an element-wise Hadamard product $(A \circ B)_{kj} = A_{kj}B_{kj}$. The paper shows that H is a positive definite matrix so we can approximate it as ($r = O(logNlog(1/\epsilon))$, where ϵ is desired precision):

$$H \approx \sum_{s=1}^{r} a_s \ell_s \ell_s^\top \tag{2.73}$$

ℓ_s is a vector that can be evaluated using pivoted Cholesky algorithm provided in the paper. The final expansion approximation:

$$R_T^P v \approx \sum_{s=1}^{r} a_s D diag(\ell_s) T diag(\ell_s) v \tag{2.74}$$

can be computed with the help of FFT, as multiplication of T with a vector is essentially a convolution.

The advantage of this approach is straightforward structure (will be easier to implement than the hierarchical decomposition), low memory requirements ($O(NlogN)$, machine-precision accuracy and low arithmetic cost after pivoted Cholesky algorithm precomputation - $r = O(logNlog(1/\epsilon))$ FFTs. I will implement it first and see which part takes more time - order connection or Chebyshev-Legendre transform, then either algorithms can be improved.

Another small benefit of the Toeplitz–Hankel approach is that it can be generalized for some other Jacobi–Jacobi transforms in the future, where the explicit representation is also known.

2.3 JWT and ALT implementation comparison

Now we will compare memory requirements, algorithms needed and accuracy of both (quadrature and order connection) approaches for both Jones-Worland and Associated Legendre polynomials.

2.3.1 Memory requirements

To calculate integrals with a quadrature, we need to store grid points and weights (for ALT) - N_g for each order n up to N. In addition to this, we need to store this for each harmonic degree l up to L[7]:

$$S_q \sim LN_gN = O\left(N^3\right) \tag{2.75}$$

Memory requirements can be lowered to $O\left(N^2\right)$ if we start to calculate the coefficients used in recurrence relations on the fly, which is the approach used in[9].

For polynomial order connection, we only need to store matrices$\mathbf{M}_i$ and$\mathbf{V}_i$, each consisting only of two diagonals. Their sizes range from N to N_T, and we have $N_T - N \sim N$ of these matrices. For ALT, we also need $O(N^2)$ storage to keep 2.65:

$$S_{por} \sim O(N^2) \tag{2.76}$$

For large N, the polynomial order connection approach should produce substantial memory savings over the conventional quadrature approach implementation without additional on-the-fly coefficients computations.

2.3.2 Accuracy

In this section we will discuss in detail the Jones-Worland polynomials accuracy scaling for the polynomial order connection approach, as ALT with it is currently in development. The accuracy reasons were the main consideration for actually using the polynomial order connection approach to Jones-Worland polynomials. By construction, these polynomials 2.1 have a prefactor r^l, that offsets the natural behavior of $P_n^{(-1/2,l-1/2)}(2r^2-1) \to r^{-l}$. Calculation of them separately (which is done in quadrature) for high l yields underflow in r^l and overflow in Jacobi polynomial already at $L = 125$, as shown in [7].

Overall, the error scaling will be defined by the most imprecise algorithm used in the computation. For polynomial order connection, the initial order connection sequence of $\mathbf{M}_i$ and$\mathbf{V}_i$ operators have constant accuracy across all Chebyshev nodes and an accuracy scaling of $O(\sqrt{N})$ for a polynomial order $l = 2N$.

The DCT has the same error scaling properties as FFT, hence $O(logN)$ scaling. Essentially, it stays on the order of machine precision for the target systems of $N \sim 10^3$.

For a Chebyshev-Legendre connection matrix 2.65, from [10] we expect that accuracy will scale not worse than $O(\sqrt{N})$. The tests of Slevinsky's code during last years retreat verified that this is correct, if the coefficients are precomputed with sufficient accuracy.

To summarize, the polynomial order connection approach is expected to have an $O(\sqrt{N})$ error scaling. In contrast, state of the art quadrature approach implementation for the Associated Legendre Transform exhibits $O(N)$ accuracy scaling[9].

3 Runtime code optimization platform

The rapid development and huge success of the GPU market in recent years provided the market with extremely fast parallel machines, which now displace CPUs as the main source of compute power in modern HPC clusters. As a result of the second year of this PhD, we will present multiple novel algorithms, optimized for modern supercomputers. They have been implemented as a part of new code generation platform, design of which has been finalized since the research plan proposal defence.

3.1 Platform for GPU code generation

While GPUs show extraordinary performance, they are harder to program for compared to regular CPUs. Firstly, the algorithms must be optimized for SIMT (single instruction, multiple threads) execution, which is the soul of the GPU model. Secondly, rapid GPU development makes all vendor architectures (Nvidia, AMD, Intel) largely different. Below we will list some examples:

- Shared memory size. If your code uses the full 64KB of shared memory on AMD MI250X, it won't be optimal on Nvidia H100 which has 256KB of it. The same goes for L2/L3 cache size.
- Warp size. If your code uses 32 core warps (usual on Nvidia), it will not work or work inefficiently for 64 core warps of AMD CDNA.
- Coalesced memory. For Nvidia and AMD we have to coalesce 32-byte memory requests, for Intel, it is equal to 64 bytes. Multiple of these requests are then combined into 128-byte transactions to the chip. However, this is a big simplification. There exist undocumented issues for Nvidia and AMD, when bandwidth drops when code tries to do distant, but coalesced memory accesses, i.e. data are still grouped in 32-byte transactions but target addresses with $> 2\hat{}18$ bytes between them (and the bigger the distance - the bigger the drop). The load-store unit (LSU) can not group 32-byte transactions. It is possible to regain full bandwidth by switching to 128-byte memory coalescing on Nvidia, while on AMD memory management follows different rules and these requests just happen to hit the same physical memory pin and thus be serialized. All this information is usually never shared by vendors, so programmers have to find solutions by trial and error.
- Compute unit composition. Some compute units have special function units that can evaluate single precision sines and cosines in one cycle (AMD and Nvidia). On Intel GPUs, they are often imprecise. The ratio between FP32 and FP64 units also imposes limitations on the algorithm - compute units with low FP64 core count will benefit from data precomputation in the lookup tables more. Some compute units have tensor cores for accelerated matrix products.
- There are many other nuanced differences (like the number of supported by hardware queues/streams, support for CUDA MPS execution model, register spilling control) which result in some complicated kernels being unpredictable in performance before you try. So having an option to tweak some of its parameters loosely related to hardware is a huge benefit.

This brings us to the third main obstacle: all vendors have their own computing platforms. CUDA for Nvidia, HIP for AMD, Level Zero for Intel, Metal for Apple. Nvidia has entered the market of HPC way in advance of everybody else and developed and optimized their closed-source software stack for their GPUs which has libraries, profiling tools, debugger, etc. This works well for them because all modern codebases turned out to be dependent on solutions provided by Nvidia, which made usage of other vendor GPUs very hard - AMD had to mimic Nvidia's efforts in a shorter time and create their own collection of similar primitives and tools, which they call ROCm. While they release it as open-source, its performance is not on par with Nvidia's and it is once again - optimized only for their GPUs. Intel has yet to release their own HPC GPUs, but they as well make their own software stack which they call Level Zero.

To facilitate the development of code for all these different in details but similar in SIMD nature GPUs, I used all the knowledge obtained via numerous experiments with VkFFT to generalize GPU programming as much as possible - and create a unified platform for GPU code generation.

3.2 Platform structure

The project structure is split into four main parts:

- Application manager. We use a container-based model of management - the user fills out the configuration struct and then initializes the application with only one initializer function. So this part of the platform performs initial configuration management, pass of the parameters to the corresponding to the particular subprogram plan initializer, binary load/store for future reuse and resources deallocation. Here we have application dispatch handlers, which abstract the actual application-dependent code dispatch handlers in the same way as we have one function for initialization.
- Plan manager. After configuration propagation, the plan manager performs extensive configuration of parameters for a particular task code generation (like a Jones-Worland $\mathbf{M}_i$ and$\mathbf{V}_i$ step). It also performs general API-dependent parameter initialization for supported backends. The plan manager also performs code compilation using supported by the target API runtime compiler (NVRTC by Nvidia CUDA, HIPRTC by AMD HIP, glslang by Vulkan, native OpenCL, clang and llvm-spirv for Intel Level Zero). It manages additional memory allocations and backend-specific memory management. Finally, it has plan-specific dispatch routines and a plan resource deallocation manager.
- Code manager. This part of the platform is solely responsible for GPU code generation. It takes a special input configuration structure created from user configuration and API-derived parameters and produces a char array containing code to be executed. The code manager has a block structure that developers can use. This allows reusing parts of code between kernels. The block structure has the following designations:
  - Level 2 kernels: problem-specific kernels and layout configuration. These functions are called by the plan manager and they are intended to have a clear description of what is going on in the function. There is no pure code here - only calls to append lower-level functions code to the kernel code and additional inlining constants configuration that can be performed by the propagated from the plan manager information. This kernel also performs calls to additional kernel configuration routines like shared memory and register inlining inside a kernel and overall kernel decoration (like calls to inlining structure definitions, calls to kernel name inlining) - these routines are defined in lower kernels levels.
  - Level 1 kernels: less abstract simple routine kernels, like matrix-vector products, PCR solvers, copy kernels, radix components of FFT, read/write managers. All of them must have a specified structure so they can be reused, for example, considering all input data to be in shared memory with a specific layout - then level 2 kernels will have a simple job of providing data to level 1 kernels in a specified manner and get the output in an also specified manner.
  - Level 0 kernels: memory management kernels. Here we have the absolute simplest kernel blocks, like memory data transfer request calls, synchronization, inlining of register/shared memory initialization code, etc. Many APIs have their own definitions for particular math operations, so we can define functions for inlining additions multiplications and other basic arithmetics here as well. The target of this level is to provide level 1 and 2 kernels with an easy way to reuse things that can be generalized between the widest range of math algorithms.

  All of the different level kernels have full access to the target GPU hardware parameters - hence (depending on the quality of implemented kernels) it is possible to autotune the algorithm at run-time and simplify the compiler job as much as possible. This involves loop unrolling, constants inlining, etc. Having an additional level (level 1) dedicated to the simple algorithms allows for kernel merging.
- Structs definitions. Here we define all structs used by the platform. It also contains some helper structs that can be used to manage GPU with different APIs and collect generated applications in a library struct for easier access through a map key (if the user's code uses many applications, this turns out to be extremely helpful).

3.3 Container abstractions

The main design improvement that enables the platform functionality and greatly expands its usability are container abstractions. The code generator now operates on special data containers (implemented as C-unions), that can hold

integer/float/complex numbers either as known during the plan creation values or as strings of variable names - no more sprintf calls with big chunks of unreadable code.

An example operation that uses containers is a MUL operation that performs a multiplication A equals B multiplied by C. If all containers have known values, A can be precomputed during plan creation. If A , B and C are, for example, register names, we print to the kernel an operation of multiplication. Each operation like MUL has its own representation for all supported APIs in the vkSolve_MathUtils.h file. An important distinction is that it is not a full compiler, but a simple tool that unifies syntaxis of different APIs and supports JIT optimizations. However, as the code manager outputs code that resembles binary, one of the future exploration possibilities it may be interesting to try to make level 0 and 1 kernels output assembly without additional compilation by the plan manager.

Usage of container abstractions brings exciting opportunities to code creation. Behaving as multi-API templates it will be possible to implement double-double FP128 mathematics support on GPU without modifying any of the already written algorithm code. Another big benefit is automatic type casting between precisions and option to perform precomputation in higher precision than the GPU code executes in.

I have also explored other ways to implement these container abstractions - including switching to C++ from C99 and using templates and operator overloading, however this turned out to be just a syntactic sugar, as if it did not reduce number of lines of code and did not increase readibility. So the code is still C with almost instant compilations as a bonus. All this is subject to change in the future if people present convincing arguments.

4 Implemented GPU algorithms

In this section the major body of work done in the second year will be presented.

4.1 VkFFT, an efficient GPU-accelerated multidimensional Fast Fourier Transform library

At the end of last year a paper on VkFFT has been finally published out in IEEE Open Access journal [11]. Before that a giant poster (3A0) covering most of its contents was presented at SC22 conference in Dallas, US. It is worth noting the following advances made in VkFFT:

4.1.1 Rader's FFT algorithm implementation

The parallelization of Cooley-Tukey and Stockham algorithms is usually done with a single thread (either CPU or GPU) responsible for a single N_1 radix kernel. Then we can have $N_2 = N \div N_1$ threads working at the same time. To achieve good occupancy on GPUs, N_2 has to be big (at least the number of available cores). However, this is often not the case when N_1 is a big prime (for example, when the full sequence length N is a prime). Another issue is that the generation of optimized radix kernels for big primes is a non-trivial task in itself. Luckily, there exist two FFT algorithms that solve this issue: Rader's FFT algorithm and Bluestein's FFT algorithm. We will focus on Rader's FFT algorithm, as Bluestein's FFT algorithm has been implemented before the research plan defence. You can find the Rader's FFT algorithm description below and on Fig. 4.1.

Rader's FFT algorithm computes the FFT of a prime-sized length P as a $P-1$ length cyclic convolution (Fig. 1). The main idea behind the algorithm is that residuals $[1, P-1]$ form a multiplicative group of integers modulo P with a generator g. Below, all generator powers are calculated modulo P. Rader then reformulates the FFT in question (reorder steps in Fig. 1) as:

$$X_0 = \sum_{l=0}^{P-1} x_l$$

$$X_{g^k} = x_0 + \sum_{l=0}^{P-2} x_{g^l} e^{-\frac{2\pi i}{N} g^{(l+k)}},\ k \in [0, P-2]$$

The second summation is $P-1$ length cyclic convolution. It can be calculated in two ways - as a direct multiplication with a precomputed kernel, or by using the convolution theorem ($P-1$ FFT to IFFT steps in Fig. 1) for the following two sequences:

$$\mathrm{DFT}\{f * h\} = \mathrm{DFT}\{f\} \cdot \mathrm{DFT}\{h\}$$

$$a_n = x_{g^l}$$

$$b_n = e^{-\frac{2\pi i}{N} g^{(l+k)}}$$

The b_n sequence and its DFT can be precomputed. The direct multiplication approach does not scale well with increasing P as it has $O(N^2)$ complexity (however it is also implemented in VkFFT and works well for primes up to 100). The convolution theorem version of Rader's algorithm has $O(NlogN)$ complexity and works well for any prime, if it fits in the shared memory of the GPU - primes of sizes near 10000 for FP32. The Rader's FFT algorithm has another limitation - we rely on being able to compute a $P-1$ length sequence FFT with a regular Stockham algorithm approach, which is not the case for Sophie-Germain safe primes (numbers like 47, 59, 83) and sequences divisible by them, so we use regular Bluestein's algorithm if we encounter Sophie-Germain safe primes bigger than 100 in the sequence decomposition.

VkFFT is the only GPU FFT codebase that uses Rader's FFT algorithm - the only close competitior is Nvidia's cuFFT and it only uses Rader's algorithm for primes up to 127 and implements it as a direct multiplication. On Fig.4.2 and Fig.4.3 the benchmarks plots are shown that demonstrate clear superiority of the implemented algorithm. Pictures from [11].

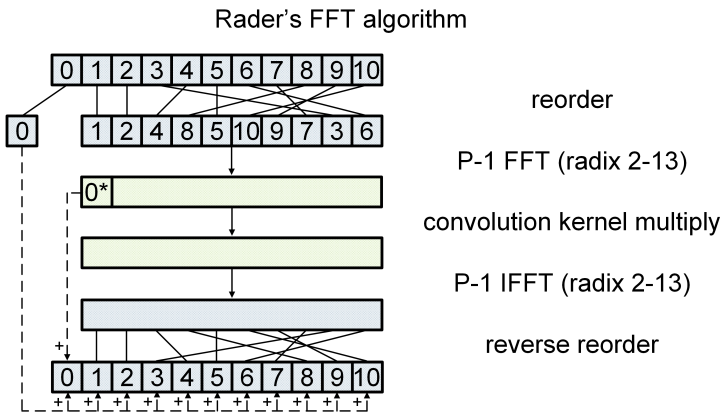

Figure 4.1: Rader's FFT algorithm step-by-step description. This figure shows how an FFT of length 11 can be computed as a convolution of length 10. Convolution is performed with the help of the convolution theorem (light green), which can be done efficiently with the Stockham algorithm for primes from 2 to 13.

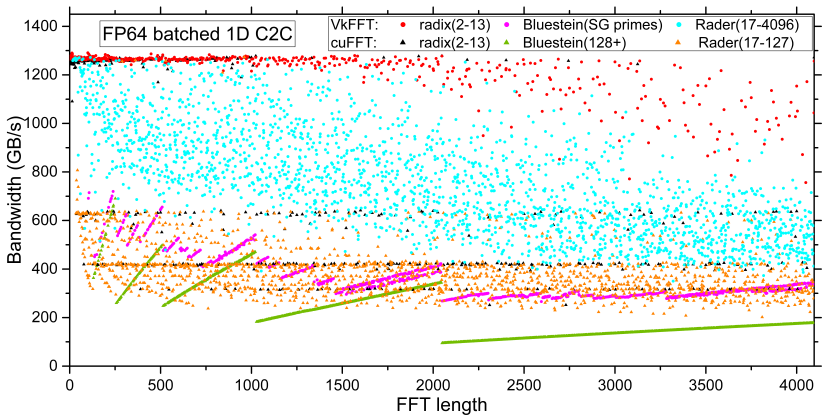

Figure 4.2: Benchmark of Nvidia A100 GPU with VkFFT (circles) and cuFFT (triangles) in batched 1D double-precision FFT+IFFT computations. Different colors represent different algorithms used for this particular sequence size: red and black are radix decomposition, magenta and green are Blustein's algorithm, cyan and orange are Rader's algorithm.

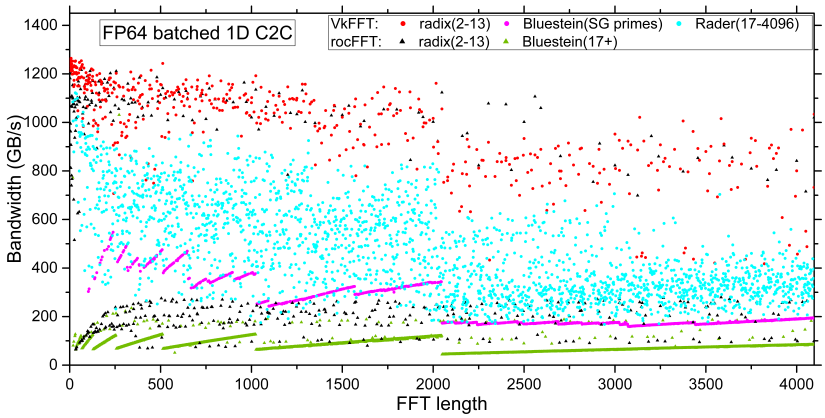

Figure 4.3: Benchmark of AMD MI250 GPU with VkFFT (circles) and rocFFT (triangles) in batched 1D double-precision FFT+IFFT computations. Different colors represent different algorithms used for this particular sequence size: red and black are radix decomposition, magenta and green are Blustein's algorithm, cyan is Rader's algorithm.

4.1.2 VkFFT switch to the platform design

Original VkFFT codebase was a 43k lines of code single header file with unreadable (to others) sprintf calls and a lot of code duplication - especially in the read/write stage of data management of GPU kernels. The major undertaking was done to split the code according to the platform design. It has been marked as completed in April 2023 with the release of VkFFT development version 1.3 in time for my IWOCL 2023 30-minute talk about the possibilities the code generation platform opens up [12].

The read/write stage of data management of GPU kernels reorganisation deserves additional mention. The memory management is extremely important in GPU programming as usually the GPU can make 50-100 math operations per one memory transaction from the GPU device memory. So, the code for this initially was 10k lines in total, consisting of separately optimized algorithms for all different types of complex and real transforms and types of memory accesses (strided and nonstrided) and it was the hardest part of code to bring new features in. Now it has been merged into an optimal 1.5k LOC efficient algorithm that is also used in all other QuICC kernels.

Overall, the VkFFT is now a stable codebase that is used by many big projects (like GROMACS molecular dynamics code) and people are now even making contributions to it.

4.2 Jones-Worland and Associated Legendre Transforms in QuICC

To perform the polynomial order connection algorithm, we need matrix multiplication (by a matrix consisting of two diagonals), a bidiagonal matrix solver (for Jones-Worland), a diagonal + second sub/super diagonal solver (for ALT), efficient matrix multiplication for a connection matrix 2.65 and Discrete Cosine Transform implementation.

4.2.1 Theoretical algorithms description

- Matrix multiplication by a matrix consisting of two diagonal can be done in $O(N)$ steps and is easy to parallelize - each core takes two matrix values and two vector values, multiplies and adds them and then writes to the output location - there are no data hazards as no core writes to the location other cores read from. I have also implemented a regular banded matrix multiplication algorithm that sometimes is required in QuICC.
- A bidiagonal matrix solver and a diagonal + second sub/super diagonal solver. Both of these tasks can be done in $O(N)$ steps with a simple backward substitution. However, the backward substitution algorithm is not parallel by its nature - each consecutive step uses the backward substitution results from the previous step. Fortunately, parallel versions of required solvers exist. I have implemented a bidiagonal solver based on a parallel cyclic reduction algorithm for solving the tridiagonal system in the subsection. It has an $O(Nlog_2N)$ complexity, which is balanced by the fact that work can be distributed among N processors equally[16]. A diagonal + second sub/super diagonal solver is essentially a bidiagonal solver of two independent systems of size $\frac{N}{2}$, so no new algorithm is needed in this case (even though a pentadiagonal modification of the tridiagonal solver can be trivially constructed[2]). Below the formal description of the PCR algorithm is given

$$\begin{array}{ccccccc} a_{i-1}x_{i-2} & +b_{i-1}x_{i-1}+ & c_{i-1}x_i & & & = & y_{i-1} \\ & a_ix_{i-1} & +b_ix_i & +c_ix_{i+1} & & = & y_i \\ & & a_{i+1}x_i & +b_{i+1}x_{i+1} & +c_{i+1}x_{i+2} & = & y_{i+1} \end{array} \tag{4.1}$$

by using the multiplier $\frac{a_i}{b_{i-1}}$ for the y_{i-1} equation and $\frac{c_i}{b_{i+1}}$ for the y_{i+1} equation and subtracting them from y_i equation, we get:

$$-\frac{a_ia_{i-1}}{b_{i-1}}x_{i-2} + 0 + (b_i - \frac{a_ic_{i-1}}{b_{i-1}} - \frac{c_ia_{i+1}}{b_{i+1}})x_i + 0 - \frac{c_ic_{i+1}}{b_{i+1}}x_{i+2} = -\frac{a_i}{b_{i-1}}y_{i-1} + y_i - \frac{c_i}{b_{i+1}}y_{i+1} \tag{4.2}$$

This step converts the system from banded tridiagonal form to the main diagonal + second sub/super diagonal. The systems of odd-indexed and even-indexed equations form two tridiagonal systems of size $\frac{N}{2}$. At the next step, each equation i will require updated coefficients from equations $i-2$ and $i+2$, then $i-4$ and $i+4$ and so on. If the coefficient index is outside the matrix, we use 0 instead. After $[log_2N]$steps, all equations are reduced to the form, solvable by simple division. Fig.4.4 shows the access pattern of PCR for all stages.

In case of JWT and ALT, we only have two diagonals, meaning that one of $\frac{a_i}{b_{i-1}}$ or $\frac{c_i}{b_{i+1}}$ is zero, which allows to reduce the number of memory transfers and calculations. Another optimization is that decomposing the solve matrix as a multiplication of a diagonal matrix with main diagonal elements and a scaled bidiagonal matrix with ones in the main diagonal, allows to reduce the memory transfers as $(b_i - \frac{a_ic_{i-1}}{b_{i-1}} - \frac{c_ia_{i+1}}{b_{i+1}}) = 1$ and we do not have to perform any divisions (expensive operation on the GPU) in the algorithm. The diagonal multiplier can then be inverted and merged with the matrix multiplication step during precomputation.

- Efficient matrix multiplication for a connection matrix 2.65. While direct multiplication is possible in this case as well with an $O(N^2)$ complexity, the structure of matrices 2.65 allows for an off-diagonal compression with the Fast Multipole Method with an $O(NlogN)$ complexity[1, 10].
- Discrete Cosine Transform can be computed efficiently by mapping it to the Fourier transform of a similar length, hence the $O(NlogN)$ complexity. It has been implemented as a part of VkFFT during the first year of the PhD and so far is the only general DCT algorithm implementation for GPUs.

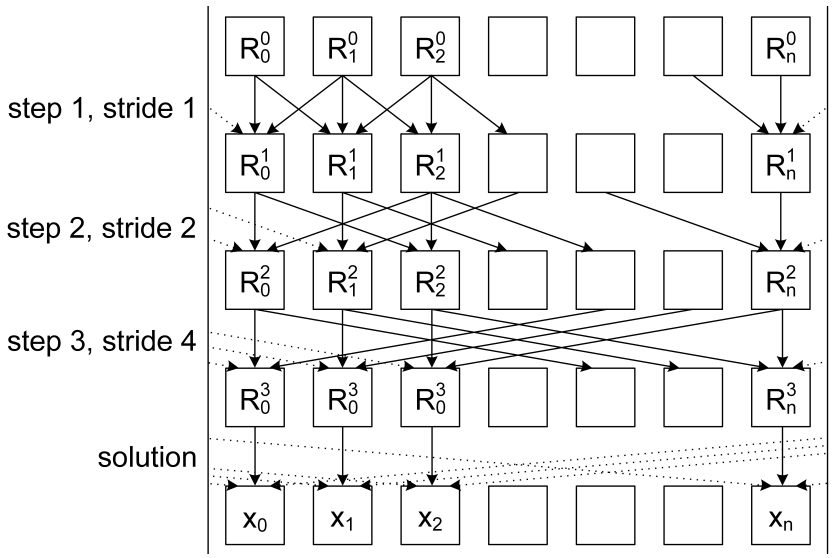

Figure 4.4: Memory access pattern of the PCR algorithm. The algorithm reduces one 7-equation system to two 4-equation systems in step 1, from two 4-equation systems to four 2-equation systems in step 2, from four 2-equation systems to eight 1-equation systems in step 3, and finally solves eight 1-equation systems. At each step all cores are fully utilized, a total number of steps is equal to $log_2 8$.

4.2.2 GPU PCR algorithm description

With the help of devoped platform it was possible to create and utilize a novel GPU programming approach - single warp kernels. Warp – the basic unit of execution, a collection of threads (32 on Nvidia, 64 on AMD), that are executed simultaneously on a single streaming multiprocessor (collection of GPU cores, an analogy of a CPU core in terms of power). We propose a new GPU code design paradigm:

- One warp per kernel per solved system design.
- Number of kernels is equal to the number of solved systems.
- No shared memory (on-chip memory that is accessible and adressable by all threads in the kernel) is used – all calculations are performed in the registers. Fast data transfers between threads are done via shuffle instructions that allow to switch register ownership between threads in one warp.
- No synchronizations means low number of threads is sufficient to cover latencies. During PASC23 conference an Nvidia engineer did not believe that until I showed the code profiling and binary structure.

The warp shuffle is often used in a form of registers values shift up or done the threads indexes. In case of PCR, all shifts are powers of 2 and for shifts bigger than 32 we do not even need to transfer data, as the request targets registers inside the thread. The Fig. 4.5 shows how shuffling works on modern GPUs.

4.2.3 Standalone PCR performance verification

In the benchmark given at Fig. 4.6 we solve bidiagonal systems in FP64 with optimized PCR on Nvidia A100 and AMD MI250 GPUs for different range of batches (right-hand sides). Bandwidth is estimated as 2x system buffer size - this is the minimal transfer size as we need to get the system to and from the chip if we exclude bidiagonal matrix coefficients transfers, as they are likely to have a cache hit. If we divide this value by the kernel execution time, we can compare the bandwidth with what the GPU is capable of - 1.3 TB/s for both GPUs for simple memory copy operation. If the estimated bandwidth is close to this value, there only way of improvement will be to have multiple solves be done on chip without intermediate transfers. We will come back to this in the future work section 5. The bandiwdth can be higher than this value for small systems if they can hit in L2 cache of the GPU.

The results of this benchmark show that batch size of 10 is too small for GPU execution - GPU has more than 100

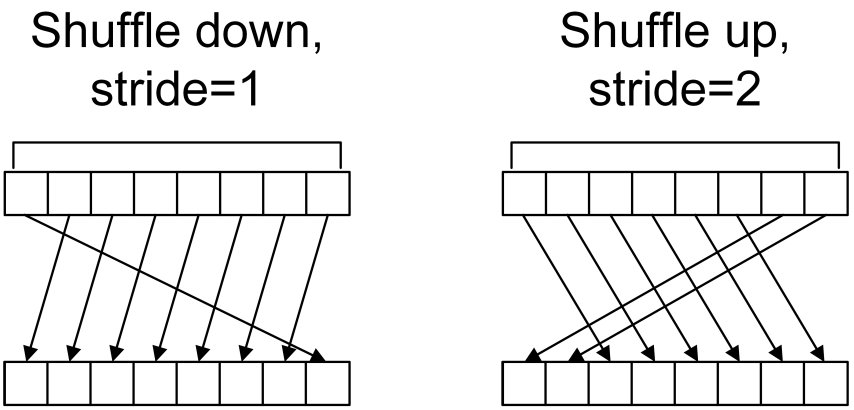

Figure 4.5: Data movements via the shuffle instructions. Each thread gets a register value of another thread in a predictable cyclic strided pattern.

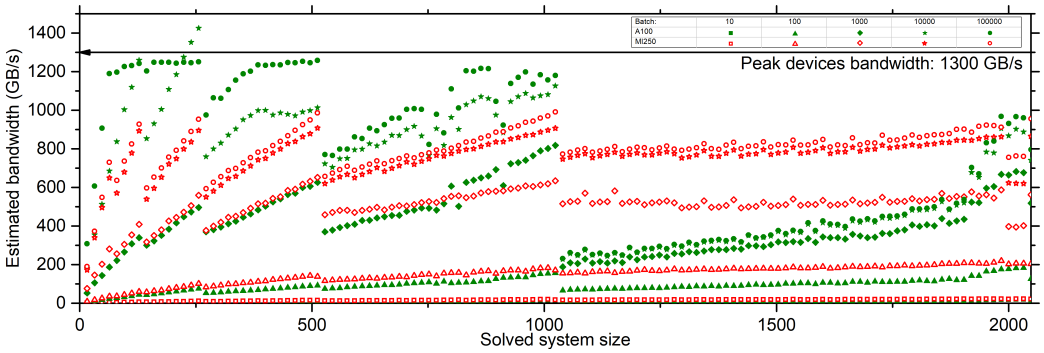

Figure 4.6: Estimated bandwidth scaling depending on the solved system dimensions and number of right-hand sides solved (shape of symbols). Results are provided for Nvidia A100 (green) and AMD MI250 (red).

streaming multiprocessors, so it is not possible to use them all here. Batch of 100 has performance comparable to CPU due to the dispatch latencies. Batches of sizes more than 1000 use 50-100% of GPU, which is a good result. For example, simple data parallel approach where a single thread solves a full system requires batching of two orders more to achieve comparable results to PCR. The warp shuffle algorithm acheves 3-4x performance improvement over the initial version of shared memory PCR presented in the first year research plan defence performance plot.

PCR algorithm solves systems as the next power of 2 with linear scaling between them meaning that it is balanced in compute/memory. For sizes > 1024, AMD benefits from having a larger warp size, as Nvidia can only use 255 registers/thread and spills some of them to the device memory. Multiple future improvements of PCR algorithm will be discussed in the section 5.

4.2.4 Standalone PCR accuracy verification

GPU PCR accuracy is compared to CPU Backward Substitution (BS) in FP64 for JW connection matrices, results are verified against FP128 backward substitution method on CPU. PCR and BS are accurate if the absolutes of the main diagonal elements are bigger than the respective sub-/super diagonal elements which is true for the JW polynomial connection matrices. Top left plot on Fig. 4.7 confirms that for fixed order, PCR and BS have almost fixed accuracy, close to machine precision (PCR is slightly worse due to it performing more operations). Top right plot confirms that for small order β, both PCR and BS experience similar accuracy drop for high (compared to order) degree polynomials – sub-/super diagonal elements become close in value to the main diagonal elements. The solution for this is to use double-double FP128 emulation which will be implemented as a data type in the code generation platform with almost no algorithm code modifications.

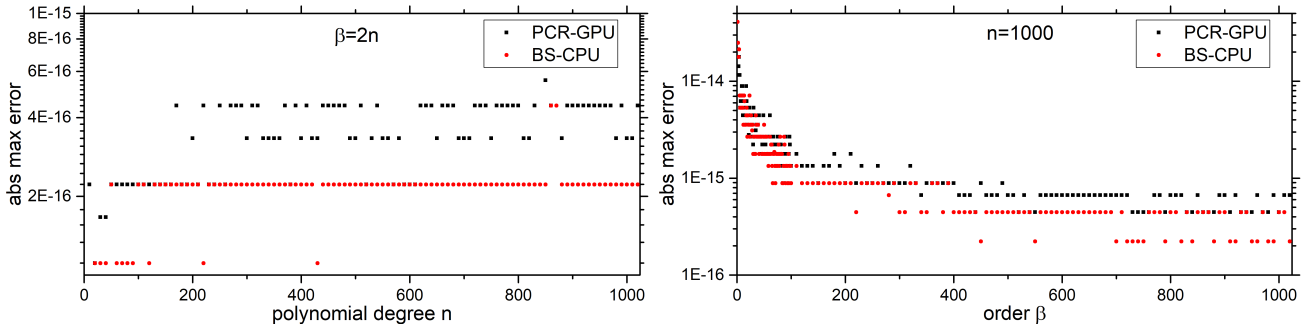

Figure 4.7: Max error across the nodes scaling for fixed order to degree ratio for different JW polynomial degrees (left). Max error scaling depending on order for fixed degree n=1000 (right).

4.2.5 Polynomial transforms usage in QuICC

The algorithms for polynomial order connection JWT were roughly implemented in QuICC with the initial version of the platform. However, new version of platform (with unions and better syntax) and improved single-warp algorithm forced the clean-up of the code, which has now been complete. The main slow down of the devlopment is that all the algorithms (matrix multiplications, scalers, data transfers, zero-padding) have to be GPU-implemented at the same time so we have only one transfer roundtrip between the CPU to the GPU, as this memory link is ~10-30 times slower than the GPU dedicated memory. The performance results will be shown on Thursday (and I will upload the working code on a separate GitHub branch for review).

The development of QuICC version of kernels pointed out one of the drawbacks of runtime generation - if there are too many parameters, the number of kernels can reach hundreds of thousands and compiling and storing each of them can take a lot of time. For this task a version of specialization constants has been implemented in the codegeneration platform. Specialization constants are numbers that can be provided to the kernel at launch - strides, offsets, scaling constants and potentially even the system size can be provided at launch (as all systems are solved as the next power of 2). This reduces the number of kernels to ~100 (potentially ~10) and reduces the compilation time almost having no affect on performance. Another option to mitigating this is generating binaries directly from the platform, but there exist no instant GPU binary interpreters, so this can be a risky undertaking (even though the generated platform code is already quite similar to the device assembly).

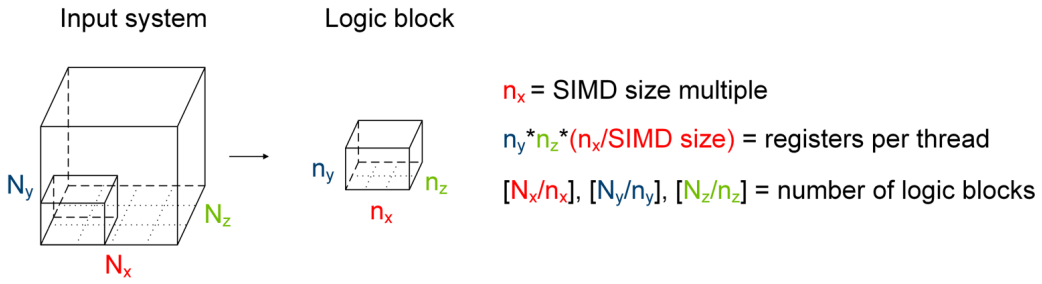

Figure 4.8: The logic block description as a single warp/ single kernel GPU partition of the solved 3D system.

4.3 Finite difference solver implemented with the single warp kernel programming approach

The outcome of the Julia course I took in the Autumn 2022 semester. The Julia programming language will not be discussed here, but I cleaned up code for a small runtime code generation platform demonstration in the IWOCL 2023 talk. A single working element in this finite difference solver is called a logic block and it can be seen on Fig. 4.8. It has a singular SIMD size group of threads dedicated to it (similarly to the PCR algorithm) and has configurable dimensions which define the workload per one kernel. Each logic block contains all respective data from the main system in its threads registers. Each thread is responsible for all data in y and z dimension slice of the logic block and if the nx is bigger than a SIMD size, every simd size + threadID x axis values. Total number of logic blocks is equal to the number of kernel dispatches. Once again, we have no synchronizations inside, but low number of threads is still sufficient to cover latencies. Configurable size of logic block, due to it being implemented in the code generation platform, allows for fine-tuning of halo effects impact on data transfers. Different logic block sizes also have different cache hit patterns, which are hard to estimate, which is another reason to have them tunable.

On Fig. 4.9 we will compare the performance of modern HPC GPUs, such as Nvidia A100 and AMD MI250 and consumer level GPU Nvidia 2080 on calculating second derivatives on each axis for three real functions respectively and then adding the result in double precision. The system is an evenly discretized cube. The benchmarks are provided for two different orders of accuracies - second (which uses two neighboring elements) an fourth (which uses four neighboring elements). Having multiple axes helps to estimate the cache effects on the performance.

On the image we see the dpendence of the estimated bandwidth (calculated as 4 system sizes (3 uploads 1 download) divided by the execution time) on the cube discretization. The bandwidth is not accurate as it does not account for additional data transfers related to halo effects on the logic blocks, however it shows the peak value as we have to always do 3 uploads of the input systems and one download of the result. Near each point you can see the best logic block dimensions for this particular configuration.

As can be seen different systems achieve best performance with different logic block configurations - this performance difference can reach 30% compared to the median solution across all results. For example, big systems such as 1024 cubed can no longer rely on L2 cache hits for z axis finite differences(it is most noticeable on Nvidia GPUs), so it is benefitial to have bigger logic block z size to have less requests.

Nvidia 2080 GPU has a very small L2 cache, so it benefits more from varying logic block size. Also it has low double precision core count, but still remains bandwidth bound (even though compute pipeline usage jumps from 20% to 80% compared to FP32, as reported by the ncu profiler).

Overall, all GPUs achieve up to 80% ratio of estimated bandwidth to the theoretical bandiwdth for second order scheme and 60% for fourth order. If we use the more sophisticated tools for profiling, like Nvidia ncu profiler, we will see that actual memory bus utilization is near 100% due to the cache misses related to halo effects of logic blocks.

While this code is done as a simple demonstration and proof of concept that single warp GPU programming works, maybe it can be useful to some people in the future.

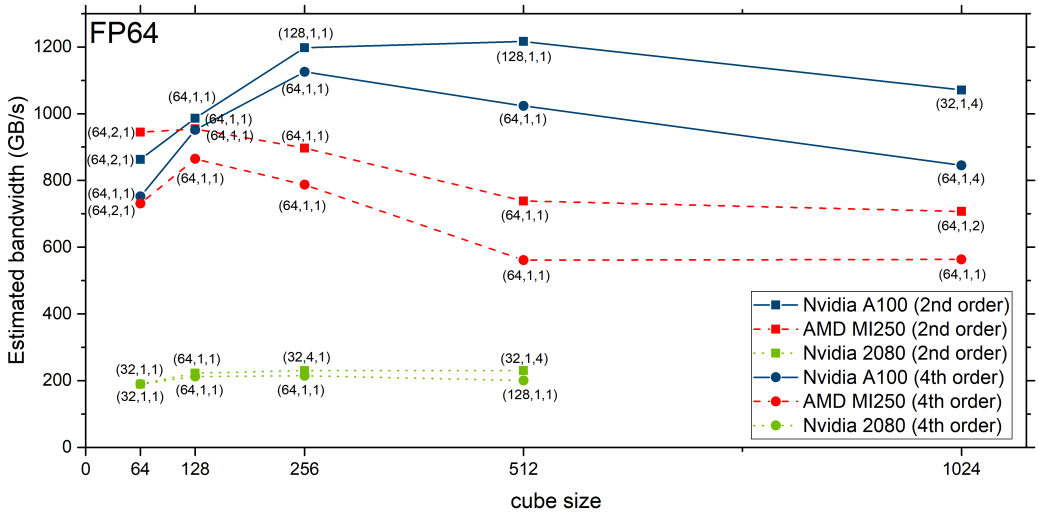

Figure 4.9: The performance results of solving cubic systems in FP64 on GPUs with HPC and consumer level GPUs.

4.4 Nonuniform Fast Fourier Transforms

Following the professor Andersson's lecture, a new algorithm has been developed to compute the spreading function convolution, which is the most time-consuming step of NUFFT. Let's briefly define the NUFFT algorithm of type I (uniform spatial sampling, nonuniform frequency sampling):

$$F_n = \sum_{j=1}^{N} \rho_j e^{-inx_j}, n = -\frac{N}{2}, \ldots, \frac{N}{2}, x_j \in [0, 2\pi] \tag{4.3}$$

which can also be expressed as exact Fourier transform of the function:

$$\rho(x) = \sum_{j=1}^{N} \rho_j \delta(x - x_j) \tag{4.4}$$

So NUFFT calculation follows the following steps:

- Convolution of 4.4 with a localized spreading function (for example, Gaussian, Fig. 4.10) and then oversampling to 2N on equispaced grid:
$$\rho_{sm}(x) = \rho(x) * g_{sm}(x) \tag{4.5}$$
- FFT of size 2N of $\rho_{sm}(x)$:
$$F_{sm}(n) = \sum_{j=1}^{N} \rho_{sm}(x) e^{-2\pi inx} \tag{4.6}$$
- Correct for the spreading step by deconvolution:
$$F_n = F_{sm} / \sum_{j=1}^{N} g_{sm}(x) e^{-2\pi inx} \tag{4.7}$$

This works due to the convolution theorem property:

$$FT(\rho_{sm}(x)) = FT(\rho(x)) \cdot FT(g_{sm}(x)) \tag{4.8}$$

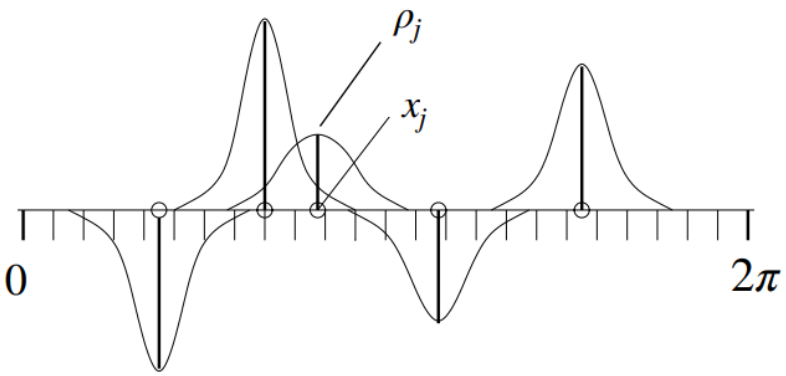

Figure 4.10: Oversampling of non-equispaced grid to the equispaced grid. For each point at the equispaced grid we add the values of the Gaussian tails from all the non-equispaced points. To improve performance, we can also introduce tunable cut off parameter limiting the tail of the Gaussian. Image from [3]

4.4.1 The single warp approach for spreading function convolution

As all other steps are $O(NlogN)$, a special care must be applied to the convolution of nonequispaced grid, as it can only be computed directly. The idea here is, once again, we compute all values of Gaussian tails in the registers of a single warp and using subgroup shifts shown in Fig. 4.5 we add values across the warp. With the help of runtime generation platform, we write a specific code for each equispaced oversampling point based on the values of x_j. This means if one of the equispaced points has M Gaussian tails reaching it, each thread in warp will compute $M/warpsize$ values and then efficiently add them in $log_2(warpsize)$ warp shuffles. If the number of Gaussians M per point is smaller than half the warp, we can calculate two points in one sequence of warp shuffles. This algorithm is highly unusual, as we have a specific codepath for every point of the oversampled to 2N grid and it is impossible to create such code without runtime generation - final code spans tens of thousands LOC. So far there have been extremely limited performance and no accuracy tests, but maybe this can be useful in the future.

5 Future work

In this section, a description on what will be the next couple months of this PhD will be given. We will merge the JWT code into QuICC in July 2023, then I will implement Chebyshev-Legendre transform for ALT calculation, as described in the document. This will be the first accurate implementation of JWT and ALT on GPUs, so this will likely warrant a paper on the topic.

The develop version of VkFFT with the new platform structure will become the main one, as the grace period of adoption for it (outlined when it was released) has passed. The Rader's algorithm may also be turned into a paper, as this is the first GPU implementation of it.

The PCR algorithm can be improved in multiple ways. First, it can be switched to PCR-CR algorithm, when we do not shuffle all tridiagonal systems until the sub/super diagonal elements become zero, but switch to backward substitution when the original system has been reduced in size to $N/warpsize$. This way the computation complexity of the algorithm will go from $O(Nlog_2N)$ to $O(N)$, as we will only need $log_2warpsize = const$ shuffle stages.

Another improvement to the PCR comes from the fact that we use fixed number of threads for all system sizes. As JWT and ALT require multiple steps of banded matrix multiplications and backsolves in a row on the same data, we can generate a single super-kernel that keeps all data in registers during all order reduction steps. Then the only data transfers needed will be the connection matrices uploads, making the kernel not bandwidth bound. Total performance gains are hard to estimate in advance, as we do not know how much computationally bound the PCR algorithm is.

It may be also good to explore the possibility of direct GPU binary generation for instant code compilation (this is required for good NUFFT speed if the grid is changing), however it is a bit out of the scope.

References

[1] Bradley K. Alpert and Vladimir Rokhlin. A fast algorithm for the evaluation of legendre expansions. *SIAM J. Sci. Comput.*, 12:158–179, 1991.

[2] Abhijit Ghosh and Chittaranjan Mishra. A parallel cyclic reduction algorithm for pentadiagonal systems with application to a convection-dominated heston pde. *SIAM Journal on Scientific Computing*, 43:C177–C202, 01 2021.

[3] Leslie Greengard. The nonuniform fft and its applications, 2020.

[4] Jens Keiner and Daniel Potts. Fast evaluation of quadrature formulae on the sphere. *Math. Comput.*, 77:397–419, 2008.

[5] Philip W. Livermore, Chris A. Jones, and Steven J. Worland. Spectral radial basis functions for full sphere computations. *J. Comput. Phys.*, 227:1209–1224, 2007.

[6] Philippe Marti and Andrew Jackson. A fully spectral methodology for magnetohydrodynamic calculations in a whole sphere. *J. Comput. Phys.*, 305:403–422, 2016.

[7] Philippe Marti and Andrew Jackson. Accurate and efficient jones-worland spectral transforms for planetary applications. *Proceedings of the Platform for Advanced Scientific Computing Conference*, 2021.

[8] Sheehan Olver, Richard M. Slevinsky, and Alex Townsend. Fast algorithms using orthogonal polynomials. *Acta Numerica*, 29:573 – 699, 2020.

[9] Nathanaël Schaeffer. Efficient spherical harmonic transforms aimed at pseudospectral numerical simulations. *Geochemistry*, 14:751 – 758, 2012.

[10] Richard Mikaël Slevinsky. Fast and backward stable transforms between spherical harmonic expansions and bivariate fourier series. *Applied and Computational Harmonic Analysis*, 47(3):585–606, 2019.

[11] Dmitrii Tolmachev. Vkfft-a performant, cross-platform and open-source gpu fft library. *IEEE Access*, 11:12039–12058, 2023.

[12] Dmitrii Tolmachev. Vkfft and beyond - a platform for runtime gpu code generation. In *Proceedings of the 2023 International Workshop on OpenCL*, IWOCL '23, New York, NY, USA, 2023. Association for Computing Machinery.

[13] Alex Townsend, Marcus Webb, and Sheehan Olver. Fast polynomial transforms based on toeplitz and hankel matrices, 2016.

[14] Lloyd N. Trefethen. Is gauss quadrature better than clenshaw-curtis? *SIAM Rev.*, 50:67–87, 2008.

[15] Mark Tygert. Fast algorithms for spherical harmonic expansions, iii. *ArXiv*, abs/0910.5435, 2010.

[16] Yao Zhang, Jonathan Cohen, Andrew A. Davidson, and John D. Owens. Chapter 11 - a hybrid method for solving tridiagonal systems on the gpu. In Wen mei W. Hwu, editor, *GPU Computing Gems Jade Edition*, Applications of GPU Computing Series, pages 117–132. Morgan Kaufmann, Boston, 2012.