Indian Institute of Technology Hyderabad

Department of Materials Science and Metallurgical Engineering

Project Report On

Multi-node and Multi-GPU Scaling Study of FFTs

Shreyas A Shivapuji
B.Tech - IV Year
MS24BTNWL11001

Project Advisor: Dr. Anuj Goyal

Abstract

Density Functional Theory (DFT) simulations are computationally intensive and rely heavily on fast Fourier transforms (FFTs) to switch between real-space charge densities and reciprocal-space wavefunctions. In this project, a 3D Poisson solver, central to updating electrostatic potentials in plane-wave DFT, was isolated and chosen as a subproblem to focus on FFT performance. Four implementations of the code were written: a CPU solver using MPI+FFTW, a single-GPU solver with cuFFT, a multi-GPU (single-node) solver with cuFFT's multi-GPU extensions (cuFFTXt), and a multi-node multi-GPU solver with cuFFTMp, with the last one being untested. Prior to the implementation, the GPU-accelerated version of Quantum ESPRESSO was first set up on the SUKSHMA HPC cluster and the QE FFTXlib module was examined to understand the architecture of the QE suite. Benchmarking results demonstrate significant speedups on GPUs compared to CPUs (over $10\times$ faster for a 1024^3 grid on a single GPU) and highlight the challenges of scaling across multiple GPUs. In particular, the multi-GPU implementation showed limited acceleration due to communication overhead. This report outlines the theoretical background, implementation details, and performance analysis of the solver, laying the groundwork for integrating multi-GPU acceleration into large DFT solvers like Quantum ESPRESSO.

Contents

Chapter 1

Introduction

Modern scientific computing often involves solving large-scale physical simulations that demand high performance. Plane-wave DFT (Density Functional Theory) is one such area, widely used in computational materials science, where a significant portion of the runtime is spent performing three-dimensional FFTs. FFTs are used to transform electron density and wavefunctions between real space and reciprocal (Fourier) space. As system sizes grow, these FFT operations become a bottleneck, especially on traditional CPU-based clusters. GPU accelerators, with their massively parallel architecture, offer a promising route to speed up FFT-intensive workloads.

This project targets the acceleration of the plane-wave DFT workflow by focusing on a surrogate sub-problem: solving a 3D Poisson's equation using spectral methods (FFTs). The Poisson's Equation is used predominantly in the calculation of the potentials in a typical DFT calculation, and this solver is chosen as a representative kernel because it encapsulates the core FFT computation and vector operations that also appear in each self-consistent field iteration of DFT. By isolating this subproblem, performance studies can be done and analyzed independently of the complexities of an entire DFT code. With a major goal to leverage multi-GPU computing to achieve faster solutions and better scaling than conventional CPU implementations, the insights gained from this approach can indicate the feasibility of integrating a multi-GPU FFT solver into an existing electronic structure code (like Quantum ESPRESSO) to accelerate its calculations.

This report describes the background theory and the roles of FFTs in DFT (Chapter 2), the methodology and setup of the study (Chapter 3), details of the solver implementations (Chapter 4), and an analysis of the results obtained (Chapter 5). Finally, conclusions and future work are presented in Chapter 6, including the potential impact on computational physics applications and possible extensions of this work.

Chapter 2

Background

2.1 DFT and the role of FFT in Plane-Wave Simulations

Density Functional Theory is founded on the Hohenberg–Kohn theorems, which establish that the ground-state energy of a many-electron system is a functional of the electron density. In the Kohn-Sham formulation, the many-electron problem is reduced to a set of single-electron Schrödinger equations (Kohn–Sham equations) that are solved self-consistently. Plane-wave DFT codes expand the Kohn-Sham orbitals in a basis of plane waves, which naturally leads to heavy use of Fourier transforms. Key operations in each self-consistent field iteration include converting the electron density $\rho(\mathbf{r})$ from real space to reciprocal space and back. The spectral method utilizes FFT and yields a direct solution in $O(N \log N)$ operations (where N is the number of grid points), in contrast to a naïve real-space convolution which would be $O(N^2)$ and intractable for large grids. Thus, FFTs are indispensable in plane-wave DFT for efficiently switching between real-space and reciprocal-space representations.

In the context of Quantum ESPRESSO (QE), the FFTXlib module is responsible for performing these 3D FFTs. Historically, QE included an internal copy of FFTW (version 2) and now interfaces with FFTW3 or vendor-optimized FFT libraries for better performance. When running in parallel on multiple CPU cores (MPI tasks), QE distributes the 3D FFT grid among processes (typically using a slab decomposition where the grid is split along one dimension). Each MPI rank handles a slab of the 3D array (e.g., a contiguous chunk of planes) and performs local 1D FFTs on that slab. After computing one dimension of the 3D FFT locally, a global MPI All-to-All transpose is performed to redistribute data among ranks for the next FFT dimension. After two such transpose stages (for a full 3D FFT), each rank will have handled all necessary data for each dimension's transforms. This distributed-memory parallel FFT approach allows tackling large problems (each rank only needs to store $1/p$ of the full grid for p processes), but it introduces communication overhead from the all-to-all data exchanges.

The MPI-FFTW library (FFTW's MPI interface) provides convenient routines that

implement these transpose communications internally. The benefit of this approach is that it enables scalability to large problem sizes by splitting both data and memory across processors. The drawback, however, is that the all-to-all communication can become a bottleneck as the number of processes grows. For sufficiently large grids, the computation/communication trade-off is favorable and strong scaling can be achieved up to a point, but for smaller grids or very high process counts, parallel efficiency drops off due to communication latency and bandwidth limits. This challenge motivates exploring alternative strategies and hardware (like GPUs) to further accelerate FFT-heavy workloads.

2.2 Poisson's Equation and Spectral Solvers

The Poisson equation $\nabla^2\Phi(\mathbf{r}) = -\frac{\rho(\mathbf{r})}{\varepsilon_0}$ is a prototypical elliptic partial differential equation that appears in many areas of physics. In electrostatics, $\Phi(\mathbf{r})$ is the electric potential arising from a charge density $\rho(\mathbf{r})$. In electronic structure calculations, solving Poisson's equation for the Hartree potential is a crucial step performed at each iteration of the self-consistent field cycle. The classical approaches to solve Poisson's equation on a grid include real-space iterative solvers like multigrid or conjugate gradient, or direct linear solvers, but these can be slow for very large 3D grids or may not scale well with parallelism.

A more efficient alternative for periodic or homogenous systems is the spectral method using FFTs. By virtue of the convolution theorem, the operation of convolving the charge density with the Green's function of the Laplacian (which yields the potential) can be done in Fourier space by a simple division. Specifically, one can obtain $\Phi(\mathbf{r})$ by taking the FFT of $\rho(\mathbf{r})$ to get $\hat{\rho}(\mathbf{k})$, then for each $\mathbf{k} \neq 0$ computing $\hat{\Phi}(\mathbf{k}) = -\hat{\rho}(\mathbf{k})/|\mathbf{k}|^2$ (which applies the Green's function $-1/k^2$ in Fourier space), and finally performing an inverse FFT to transform $\hat{\Phi}(\mathbf{k})$ back to real space. This spectral Poisson solver yields the exact solution (for given boundary conditions) in only a few operations: two FFTs and a pointwise array division.

2.3 Evolution of FFTs for Parallel Architectures

The Cooley–Tukey FFT algorithm, published in 1965, demonstrated a computational complexity of $O(N \log N)$, which was a drastic improvement over the direct discrete Fourier transform's $O(N^2)$. However, achieving this ideal complexity on modern parallel architectures, especially on distributed systems, requires overcoming additional challenges. These challenges are relatively less severe on shared-memory systems, wherein multi-threaded FFT libraries use optimized plans and cache-friendly algorithms, like FFTW's planner with SIMD instructions, to maximize throughput. But, in the case of distributed-memory systems (MPI clusters), the aforementioned all-to-all communication needed for data transposition poses a main hurdle, making the optimization relatively challenging. Moreover, as cluster sizes increase, network latency and transfer

bandwidth limits also start to kick in and can significantly contribute in the stalling of these transpose steps.

Over the years, researchers have developed improved data partitioning and communication strategies to surmount these challenges. One such advanced approach is the pencil decomposition, where the 3D array is decomposed along two dimensions instead of one. In a pencil decomposition, each process gets a subarray shaped like a "pencil" (for example, of size $(N_x/p_x) \times (N_y/p_y) \times N_z$ for $p_x \times p_y$ processes). Since the transposes are performed in two smaller groups of ranks sequentially, this approach can reduce the volume of data exchanged in any single all-to-all operation at the cost of more complex data management. Another approach is to overlap communication and computation or to use hybrid parallelism (MPI + OpenMP) to reduce the communication frequency. For instance, Gao et al. (2016) proposed a dynamic load-balancing and partitioning algorithm for plane-wave DFT codes that improved strong scaling to larger node counts by carefully trading off communication overhead against computation. In essence, advances in parallel FFT algorithms primarily focus on minimizing or hiding communication, or at least on overlapping communication with simultaneous computatation.

Libraries such as `P3DFFT`, and the MPI FFT routines in `FFTW` each implement variations of these strategies. CPU-based parallel FFT implementations can scale efficiently to thousands of CPU cores, but beyond a certain scale the global transpose becomes a limiting factor. This observation sets the stage for GPU acceleration and newer communication-avoiding techniques (like those employed by cuFFTMp) to push performance further. By offloading FFTs to GPUs and using more advanced communication methods, one can potentially alleviate the bottlenecks that limit CPU-based scaling.

2.4 GPU Architecture and CUDA Programming Model

Graphics Processing Units (GPUs) are specialized processors designed to execute a very large number of threads in parallel. Unlike a CPU, which may have a few dozen high-performance cores optimized for sequential processing, a modern GPU contains thousands of smaller, throughput-oriented cores optimized for throughput on data-parallel tasks. For example, NVIDIA's Ampere architecture GPU (A100) contains 6912 CUDA cores, and the newer Hopper (H100) has over 14,000 cores. GPUs achieve massive parallelism by using a Single-Instruction Multiple-Thread (SIMT) execution model: groups of 32 threads (a warp in NVIDIA terminology) execute the same instruction in lockstep on different data. The hardware is designed to context-switch between warps efficiently, which helps hide latency (for instance, when some warps wait for memory, others can execute). GPUs also have a memory hierarchy tailored for high bandwidth. They feature on-device global memory (HBM2 or HBM2e) with hundreds of GB/s of throughput, as well as on-chip shared memory (user-managed cache) and registers for each multiprocessor. These characteristics make GPUs extremely well-suited for workloads like FFTs that involve repetitive arithmetic on large arrays – the GPU can

perform many "butterfly" operations concurrently, and its memory system can feed data to the compute cores at a high rate.

Historically, GPUs were exclusively designed for realistic graphic computations in video games. But with the advent of GPGPU paradigms and NVIDIA's CUDA framework, which is a parallel programming framework that allows developers to write general-purpose GPU (GPGPU) programs in C/C++ (and other languages with bindings), GPUs are being leveraged for the purpose of scientific computing as well. In the CUDA model, the CPU (host) launches kernels (functions) to be executed on the GPU (device), and manages data transfers between host memory and device memory. Efficient CUDA programs maximize occupancy (keeping thousands of GPU threads busy) and optimize memory access patterns (coalesced memory accesses to utilize full bandwidth). NVIDIA provides highly optimized libraries for common operations, such as cuBLAS for linear algebra and cuFFT for Fast Fourier Transforms, which leverage these low-level hardware optimizations internally.

A key aspect of GPU computing in HPC is overlapping computation with communication. For instance, while one batch of data is being processed on the GPU, the CPU (or another CUDA stream on the GPU) can transfer the next batch of data or perform other computations concurrently. The CUDA streams and events API enable this kind of pipeline, which is crucial for hiding data transfer latency behind computation. In the context of distributed FFTs on GPUs, one ideally wants to overlap the inter-GPU communication with the local FFT computations to minimize idle time.

2.5 GPU-Accelerated FFT Libraries (cuFFT and cuFFTMp)

cuFFT is NVIDIA's FFT library analogous to FFTW on CPUs. It provides functions to compute 1D, 2D, and 3D FFTs on data residing in GPU memory. Internally, cuFFT is optimized for the GPU architecture, breaking the FFT into batches that fit into fast shared memory and using parallel matrix transpose algorithms for multi-dimensional transforms. On a single GPU, cuFFT can often outperform a multi-core CPU running FFTW by an order of magnitude, thanks to the GPU's higher memory bandwidth and parallelism. Recent versions of cuFFT also support multi-GPU transforms on a single node (via the Xt API): one can create a cuFFT plan that uses multiple GPUs to partition the work. The cuFFTXt extension handles the data distribution (e.g., dividing the volume into slices) and uses peer-to-peer memory access over NVLink or PCIe to perform the necessary data exchanges between GPUs. In the context of this work, cuFFTXt interfaces such as `cufftXtPlanMany`/`cufftXtExec` were used to spread a 3D FFT across up to 2 GPUs in a node.

While cuFFTXt addresses multi-GPU on one node, cuFFTMp is a relatively newer and advanced library that enables multi-node, multi-GPU FFT computations. Announced in early 2022, cuFFTMp is essentially a distributed FFT library built on top of cuFFT, using a communication layer called NVSHMEM for internode data exchange. NVSHMEM provides a PGAS (Partitioned Global Address Space) model for GPUs, allowing

a kernel on one GPU to directly access memory on another GPU in the cluster via RDMA, bypassing the CPU and MPI libraries. This is critical because a traditional MPI-based all-to-all can incur high latency and CPU overhead, whereas NVSHMEM enables kernel-initiated communication, meaning the GPUs can orchestrate the transpose themselves in parallel while the CPU is idle. By leveraging NVSHMEM, cuFFTMp largely decouples FFT performance from the MPI implementation and achieves low-latency, high-bandwidth data movement across nodes. Internally, cuFFTMp still requires an MPI environment to launch ranks on each node and to bootstrap NVSHMEM (it links in an MPI plugin for startup), but after initialization, all major data exchange is done via GPU-directed communication.

The data decomposition used by cuFFTMp for a 3D FFT is a slab decomposition by default (each rank handles a contiguous slab of the 3D domain along one dimension, typically x). After the first FFT dimension is computed locally, cuFFTMp performs an all-to-all using NVSHMEM to redistribute slabs among ranks so that each rank gets the appropriate slab for the second dimension, and so on. This results in a shuffled data layout (often Y-slab) during the intermediate stage, similar to how CPU codes perform transposes. The advantage is that all communication is done through fast GPU interconnects (NVLink or InfiniBand with GPUDirect RDMA), with the transfer orchestrated in parallel by many GPU threads, thus hiding latency by sheer concurrency. cuFFTMp extends cuFFT to the multi-node realm, enabling large FFT problems to scale out on distributed memory systems. It eliminates most of the CPU involvement in communication, thereby potentially allowing overlap of communication and computation or at least reducing the communication costs significantly. For our purposes, cuFFTMp provides an ideal framework to implement a distributed Poisson solver that can utilize all GPUs on an HPC cluster for a single large 3D grid.

Chapter 3

Methodology and Study Framework

3.1 Installation and Setup of QE-GPU on SUKSHMA HPC

Throughout this project, all development and testing were conducted on the SUKSHMA high-performance computing cluster. As a preliminary step, the GPU-accelerated version of Quantum ESPRESSO (QE-GPU v7.3.1) was installed on the cluster to gain insights into the working of the software. QE-GPU requires a CUDA-capable compiler and linking against NVIDIA's CUDA libraries (cuFFT, cuBLAS, etc.). CUDA v12.3 was used, and OpenMPI was chosen as the MPI library to ensure compatibility with cuFFTMp (which uses MPI for initialization). QE was configured with GPU support (`--enable-cuda` flag and appropriate `CUDA_HOME` settings) and compiled with NVIDIA's HPC SDK compilers. FFTW3 was also installed and linked, so that QE's internal FFTXlib could use FFTW on CPUs and cuFFT on GPUs as needed.

After installation, the QE test suite was run to verify that GPU acceleration was functioning properly. QE provides timing breakdowns and prints GPU usage information when running in GPU-enabled mode; for example, it indicates if FFT operations are offloaded to GPUs. The validation was successful – the tests confirmed that FFTs and other routines were indeed utilizing the GPU. This established a known-good software environment. While exploring the different modules of the project, it was observed that QE's native GPU support (as of the version used) is mostly limited to single-node GPU usage: one MPI rank per GPU, with MPI handling communication between GPUs. There is no native support in QE for a single FFT distributed across multiple GPUs beyond what one node's MPI ranks can do (i.e., QE did not natively employ NVSHMEM or similar multi-node GPU FFT strategies). This limitation on multi-node scaling was the primary motivation to explore the integration of cuFFTMp into QE's FFT library FFTXlib.

3.2 Codebase Exploration: FFTXlib and GPU Offload in QE

To begin working on the multi-node acceleration of FFTXlib and better understand how FFTs are implemented in the existing DFT code, the source code in the official repository of the QE was explored (particularly the FFTXlib component). FFTXlib in QE provides an abstraction for 3D FFT operations, supporting both serial and parallel (MPI) transforms. It was observed that the FFTXlib typically distributes the 3D FFT grid along one dimension among MPI tasks (for example, dividing along the z-axis or x-axis). When running on CPUs, it calls routines like `fftw_plan_dft_3d` or FFTW-MPI plans for parallel transforms. In the GPU-enabled version of QE, certain steps are offloaded: QE uses GPU kernels to copy data to device memory and then calls cuFFT (through `cufftExec` plans) to perform local FFTs on each GPU. Essentially, each MPI rank (often corresponding to one GPU) handles a portion of the FFT grid and performs its local FFT; between ranks, however, QE still relies on MPI-based communication for the transpose steps. In the version examined, there was no provision for a single 3D FFT to span multiple GPUs transparently – each GPU rank handled its piece independently except for the necessary MPI exchanges.

Understanding the codebase of the QE suite involved reading through the relevant source files (such as `fft_parallel.f90`, the parallel FFT driver in QE, and related routines. To some extent, it was observed that QE's underlying FFT algorithm splits a 3D array into slabs, performs local 1D FFTs, then transposes data using MPI All-to-All. The GPU offload in QE replaces the local FFT step with cuFFT on each rank's GPU, but the transpose is still done via MPI over the CPU. This observation was important: it indicated that to use cuFFTMp (which manages the transpose on GPUs), one would have to integrate it at a lower level or outside QE's current structure. However, this integration of a new library like cuFFTMp into QE seemed rigorous and difficult to be achieved in the available timeframe, as it required significant modifications to its communication patterns and data structures, which in turn required rigorous understanding of the code structure and the file dependencies in the QE suite.

3.3 Isolating the 3D Poisson Subproblem

To evaluate multi-GPU performance in an independent and controlled way, a 3D Poisson solver was developed in C++ with MPI and CUDA. This surrogate solution used almost the same fundamentals, mathematical operations, and data distributions that QE would use for Poisson's equation and more importantly, this focused exclusively on the FFT-based Poisson equation solution, without the overhead of the rest of a DFT code. The development strategy adopted in this project was structured around three primary computational phases: (1) a forward 3D FFT of the charge density, (2) a global vector operation applying the $1/k^2$ kernel, and (3) an inverse 3D FFT to get the potential. These steps encompass the major computational components of interest (FFTs and communication) without the overhead of other parts of a DFT code and ef-

ficiently implementing these steps on multiple GPUs, across nodes, can provide insights of how much FFT performance can be improved in a best-case scenario. If the tests yielded favorable results, such as strong scaling and good efficiency, for this subproblem, the methodology and insights could subsequently be generalized and adapted to QE's codebase, particularly through targeted modifications within the FFTXlib module.

Chapter 4

Implementation Details

The following distinct implementations of the 3D Poisson solver were developed, all following the same algorithm (solve $\nabla^2\Phi = -\rho/\epsilon_o$ via forward FFT, apply $-1/k^2$ in frequency domain, then inverse FFT):

1. **CPU Implementation (MPI + FFTW)** – A parallel CPU version using MPI for domain decomposition and FFTW's MPI FFT routines for distributed 3D FFTs.

2. **Single-GPU Implementation (CUDA + cuFFT)** – A version offloading the entire problem to one GPU, using CUDA kernels and cuFFT for the FFT operations.

3. **Multi-GPU Single-Node Implementation (CUDA + cuFFTXt)** – A version that uses cuFFT's multi-GPU support (`cufftXtPlanMany`) to perform a single 3D FFT across multiple GPUs within the same node.

The pseudocode for the general spectral Poisson solver algorithm is given as Code 1, which outlines the main steps: initializing the FFT plans, populating the charge density, executing the forward FFT, applying the Poisson kernel in k-space, executing the inverse FFT, and normalizing the result. The CPU and GPU implementations are described in the subsequent sections, along with their respective pseudocodes elucidating the essential computational steps.

Code 1 Spectral Poisson Solver (General)

```
1: function PoissonSolver(ρ, Φ, N_x, N_y, N_z)
2:     (plan_f, plan_b) ← PlanFFT(N_x, N_y, N_z)
3:     ρ ← InitGaussian(ρ, N_x, N_y, N_z)
4:     ρ̂ ← FFT(ρ, plan_f)
5:     for all (i, j, k) ∈ [0..N_x − 1] × [0..N_y − 1] × [0..N_z − 1] do
6:         k_x ← WaveNum(i, N_x); k_y ← WaveNum(j, N_y); k_z ← WaveNum(k, N_z)
7:         k² ← k_x² + k_y² + k_z²
8:         if k² = 0 then
9:             ρ̂[i, j, k] ← 0
10:        else
11:            ρ̂[i, j, k] ← ρ̂[i, j, k]/k²
12:        end if
13:    end for
14:    Φ ← FFTInv(ρ̂, plan_b)
15:    Scale(Φ, 1/(N_x N_y N_z))
16: end function
```

4.1 CPU Implementation (MPI + FFTW)

In the MPI+FFTW solver, the 3D domain is partitioned into slabs along one dimension using FFTW's MPI planning interface. Each MPI process allocates memory for its local slab (size $N_x/p \times N_y \times N_z$ for p processes if split along x). FFTW provides `MPIPlanMany` which internally handles the required data transpose between processes. In our implementation, after calling MPI_Init, the FFT plan is initialized using `fftw_mpi_plan_dft_3d` (which returns forward and inverse plan objects) and the charge density array is initialized, with each rank independently setting its portion. This is followed by a FFTW call to execute the forward transform (`fftw_mpi_execute_dft`), following which the Poisson operator is applied on each local frequency-space element. Finally, an inverse transform is executed and the result is scaled by $1/(N_x N_y N_z)$ to obtain the results in the desired real space. This baseline CPU solver thus uses well-tested libraries (FFTW) for correctness and is expected to be efficient for moderate process counts, albeit limited by network communication at high scale.

Code 2 CPU Poisson Solver with MPI+ FFTW

```
1: function CPU_PoissonSolver(ρ, Φ, N_x, N_y, N_z)
2:     MPI_Init
3:     (p_f, p_b) ← MPI_PlanFFTW(N_x, N_y, N_z)
4:     DistributeDomain(MPI_COMM_WORLD,N_x, N_y, N_z)
5:     FillChargeDensity(ρ,local_slab)
6:     ρ̂ ← MPIFFT(ρ, p_f)
7:     ApplyPoissonOperator(ρ̂,local_slab)
8:     Φ ← MPIFFTInv(ρ̂, p_b)
9:     Normalize(Φ, 1/(N_x N_y N_z))
10:    MPI_Finalize
11: end function
```

4.2 Single-GPU Implementation (CUDA + cuFFT)

The single-GPU solver transfers the entire 3D data array to the GPU and performs the computation there. All the computation is devoted to a single rank or process on the GPU and there is no MPI involved in this implementation. The solver allocates device memory for the charge density and potential (of size $N_x N_y N_z$ complex values), copies the initial charge density to the GPU, and then creates a 3D FFT plan with `cufftPlan3d`. The forward FFT is executed by `cufftExecC2C` on the GPU data in-place. After the FFT, a custom CUDA kernel iterates through the frequency domain array on the GPU and applies the k-space filter (for each index computing $k_x^2 + k_y^2 + k_z^2$ and dividing the corresponding Fourier component by this number, with appropriate handling of the $k = 0$ term). Then, `cufftExecC2C` is called again for the inverse transform. A second kernel scales the result by $1/(N_x N_y N_z)$. Finally, the computed potential is copied back to the host for verification or output. All these steps are done on one GPU, so no inter-process communication is involved. This implementation tests the raw performance of a single GPU for the problem, providing a baseline for comparison against multi-CPU and multi-GPU runs.

Code 3 GPU Poisson Solver with CUDA+ cuFFT

```
1: function GPU_PoissonSolver(ρ, Φ, N_x, N_y, N_z)
2:     cudaMalloc(&(d_data),N_x N_y N_z)
3:     cudaMemcpy(d_data,ρ,H2D)
4:     plan ← cufftPlan3d(N_x, N_y, N_z)
5:     cufftExecC2C(plan,d_data,d_data,FORWARD)
6:     KernelApplyPoisson(d_data,N_x, N_y, N_z)
7:     cufftExecC2C(plan,d_data,d_data,INVERSE)
8:     KernelScale(d_data,N_x N_y N_z^{-1})
9:     cudaMemcpy(Φ,d_data,D2H)
10:    cufftDestroy(plan)
11:    cudaFree(d_data)
12: end function
```

4.3 Multi-GPU Implementation (cuFFTXt)

The multi-GPU single-node solver uses cuFFTXt, which is an extension of cuFFT that can utilize multiple GPUs on the same node for a single transform. With the usage of this API, a distributed FFT plan was created for the 3D transform across 2 GPUs (since the test node had 2 GPUs). The library automatically partitions the data (using a slab decomposition under the hood) and manages GPU-to-GPU communication via PCIe or NVLink as needed during the FFT. The code simply provides pointers to GPU memory on each device and calls cuFFTXt to execute forward and inverse transforms. The multi-GPU solver closely resembles the single-GPU case in structure, but cuFFTXt performs an additional communication while transposing the data between the GPUs after computing the FFT on each GPU's local slab, so that each can proceed with the next dimension's FFT. The advantage of this library is ease of use and minimal manual communication coding, but it is limited to GPUs within one node.

Code 4 Multi-GPU 3-D Poisson Solver with cuFFTXt

```
1: function PoissonSolve(ρ, φ, N_x, N_y, N_z, GPUs, dx, dy, dz)
2:     plan ← cufftCreate
3:     cufftXtSetGPUs(plan,|GPUs|,GPUs)
4:     cufftMakePlan3d(plan,N_x, N_y, N_z,CUFFT_C2C)
5:     desc ← cufftXtMalloc(plan, INPLACE)
6:     cufftXtMemcpy(plan,desc,ρ,Host→Device)
7:     cufftXtExecDescriptor(plan,desc,desc,Forward)
8:     for all g ∈ GPUs do
9:         (y_s, y_ℓ) ←SlabDecompose(N_y, g, |GPUs|)
10:        applyPoissonKernel(desc_g, N_x, N_y, N_z, dx, dy, dz, y_s, y_ℓ)
11:    end for
12:    cufftXtExecDescriptor(plan,desc,desc,Inverse)
13:    for all g ∈ GPUs do
14:        scaleData( desc_g, 1/(N_x N_y N_z))
15:    end for
16:    cufftXtMemcpy(plan,φ,desc,Device→Host)
17:    cufftXtFree(desc); cufftDestroy(plan)
18: end function
```

Chapter 5

Results and Analysis

All CPU runs were executed on up to 64 CPU cores (i.e., one node of dual AMD EPYC 7513 with 64 total cores), and all GPU runs were performed on `node1` of the SUKSHMA cluster, which contains 2× NVIDIA Tesla P100 GPUs (16 GB each). Thus, the multi-GPU results use 2 P100 GPUs, and the single-GPU results use one similar GPU.

The following metrics are used to evaluate parallel efficiency for comparing the performance of different implementations:

1. **Total runtime** (T_{total}) is the elapsed time to complete one full Poisson solve (forward FFT + kernel application + inverse FFT). It can be conceptually divided into computation time and communication time:

   $$T_{total} = T_{comp} + T_{comm}$$

   Here T_{comp} includes all local computations (FFT math, kernel operations) and T_{comm} includes all data exchange between processes/GPUs (MPI all-to-alls or GPU peer transfers).

2. **Speedup (S)** measures the factor by which the parallel implementation is faster than the serial or a reference implementation. If T_1 is the runtime on 1 process (or 1 GPU) and T_p is the runtime on p processes (or GPUs), then:

   $$S = \frac{T_1}{T_p}$$

3. **Parallel efficiency (E)** is the speedup normalized by the number of processes, often expressed as a percentage:

   $$E = \frac{S}{p} \times 100\%$$

   An efficiency of 100% means perfect linear scaling (e.g., doubling the number of processors halves the runtime exactly), whereas lower values indicate the presence of overheads.

Table 5.1: CPU Test 1: Strong-scaling of a 512^3 slab-decomposed FFTW transform.

Processes [n]	Forward [s]	Backward [s]	Total [s]	Mem/Proc [MB]
1	12.0591	8.3821	25.6183	8192
2	5.5776	3.5038	11.8123	4096
4	4.7350	3.0323	9.3515	2048
8	2.3900	1.9530	4.9767	1024
16	1.3260	0.9413	2.7636	512
32	0.6700	0.5202	1.4194	256
64	0.4056	0.3204	0.8511	128

5.1 CPU Baseline Results

The CPU implementation was tested on problem sizes of 512^3, 1024^3, and 2048^3 grid points, running on powers-of-two counts of MPI processes, from 1 up to 64. Figure 5.1 shows the execution time for the forward FFT stage as a function of the number of MPI processes for these three grid sizes. Similarly, Figure 5.2 shows the time for the backward FFT stage. In both plots, a logarithmic scale is used for the vertical axis to accommodate the wide range of timings.

Both forward and backward FFT times decrease as more processes are used, but with diminishing returns at higher counts. The inverse FFT is slightly faster than the forward FFT in all cases, likely because after the forward transform, data is already transposed, and due to possible cache memory. The forward and inverse FFT times significantly contribute to the CPU solver's runtime, as depicted in the Figure 5.3. From Figure 5.3, it is evident that increasing the process count reduces per-process memory (enabling larger problems per node) and initially reduces runtime, but beyond a certain point additional processes yield little to no speedup. For example, on the 1024^3 grid, the fastest time was at 32 processes; going to 64 processes actually made it slightly slower due to communication overhead.

In addition, the scaling can also be visualized through the speedup curves in Figure 5.4, which shows the achieved speedup S versus number of processes for each problem size, alongside the ideal linear speedup. It can be seen that the 512^3 case comes closest to ideal, especially up to 16 processes, and even at 64 processes it attains a speedup of ~ 30. The 1024^3 and 2048^3 cases show sublinear speedups; the orange curve peaks around 7 at 32 processes and then slightly drops, while the green reaches ~ 13 at 64 processes. The gap between these curves and the ideal line quantifies the impact of overhead.

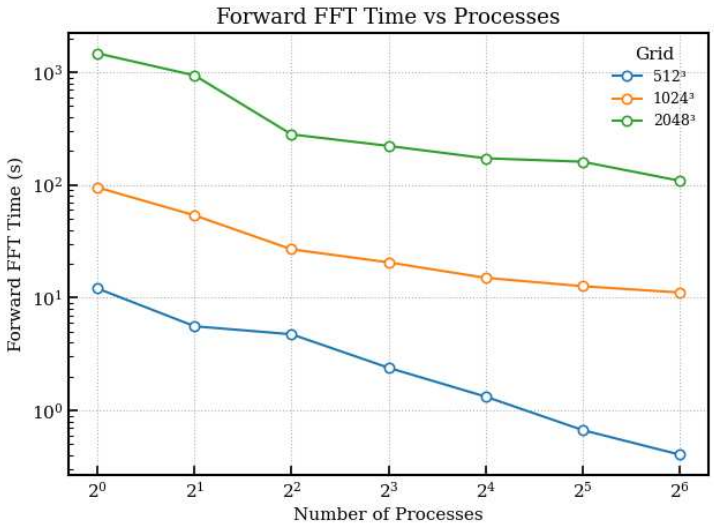

Figure 5.1: Forward FFT execution time vs. number of processes for three grid sizes in the MPI+FFTW (CPU) solver. Using more processes speeds up the FFT, but the benefit tapers off at high process counts.

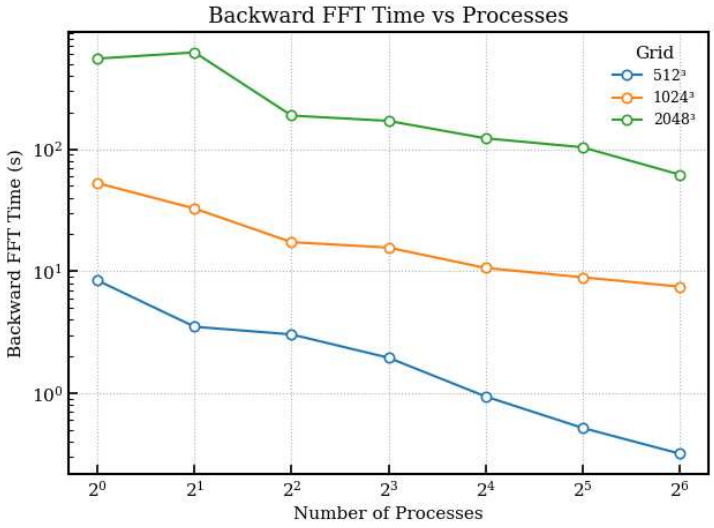

Figure 5.2: Inverse FFT execution time vs. number of processes for the same grids. The trends are similar to the forward FFT. The inverse transform is consistently a bit faster, and scaling efficiency drops for large core counts, especially on the largest grid.

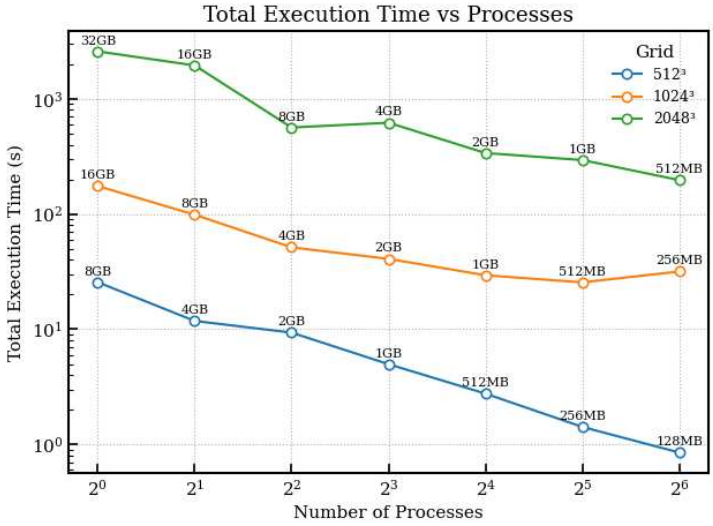

Figure 5.3: Total execution time vs. number of processes for three grid sizes. As the number of processes increases, total time generally decreases due to parallel speedup, and memory per process drops.

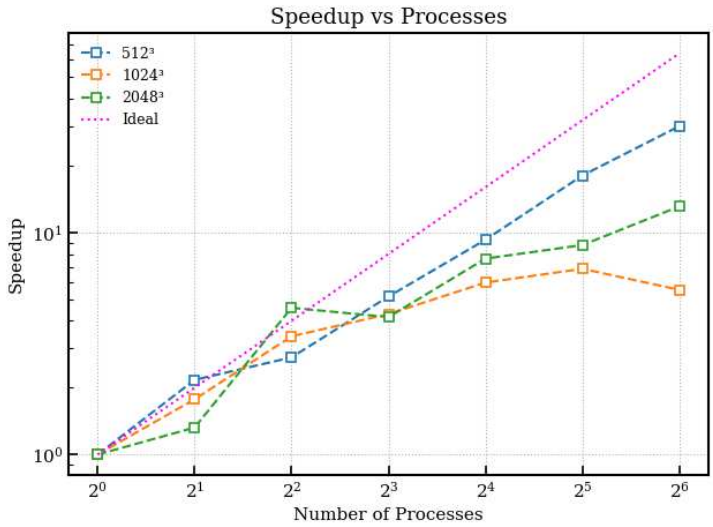

Figure 5.4: Parallel speedup vs. number of processes for the CPU (MPI+FFTW) solver. The dotted line represents ideal linear speedup ($S = p$).

Table 5.2: CPU Test 2: Strong-scaling of a 1024^3 slab-decomposed FFTW transform.

Processes [n]	Forward [s]	Backward [s]	Total [s]	Mem/Proc [MB]
1	95.0231	52.6138	174.969	16 384
2	53.7390	32.5903	98.6404	8192
4	26.8632	17.2793	51.3794	4096
8	20.5811	15.5966	40.7798	2048
16	15.0088	10.6301	29.4148	1024
32	12.6428	8.9116	25.5116	512
64	11.1230	7.4710	31.7175	256

Table 5.3: CPU Test 3: Strong-scaling of a 2048^3 slab-decomposed FFTW transform.

Processes [n]	Forward [s]	Backward [s]	Total [s]	Mem/Proc [MB]
1	1471.07	551.657	2578.01	32 768
2	937.621	620.046	1948.64	16 384
4	280.901	188.225	563.165	8192
8	222.185	170.254	620.014	4096
16	172.583	122.625	338.147	2048
32	160.832	103.524	293.800	1024
64	109.203	61.853	196.904	512

5.2 GPU Results (Single- and Multi-GPU Performance)

To examine the performance of the GPU implemented-Poisson solver on GPUs, tests were conducted on a single NVIDIA Tesla P100 GPU and on two P100 GPUs (within one node), for the same problem sizes where possible.

For the single-GPU implementation, a dramatic speedup was observed compared to the CPU runtime, even with the best-case CPU runtime (multi-cores). For instance, on a 512^3 grid, one GPU completed the Poisson solve in about 0.342 s, whereas the best CPU time was ~ 0.851 s (on 64 cores). Similarly, on a 1024^3 grid, the single GPU took 2.24 s, compared to the best CPU time of about 25.51 s (on 32 cores). These results indicate that a single P100 GPU is roughly an order of magnitude faster than 32 high-performance CPU cores for the 1024^3 problem. This is consistent with expectations, as GPUs offer massive parallelism and memory bandwidth which FFT can exploit.

In addition, the performance of the multi-GPU (2 GPUs) solver using cuFFTXt was also measured. Counterintuitively, the two GPU-setting did not outperform the single GPU-setting in the tests. For the 512^3 grid, 2 GPUs together took about 0.858 s (versus 0.342 s on one GPU) and for the 1024^3 grid, 2 GPUs took about 5.37 s (versus 2.24 s on one GPU). In both cases, using two GPUs was about 2.5× slower than using

Table 5.4: GPU Test 1: cuFFT single-GPU runs on a dual-GPU node

Dims	Forward [µs]	Backward [µs]	Total [ms]	Mem/Proc [MB]
128	36	11	103	16
256	37	10	126	128
512	53	10	342	1024
1024	40	15	2240	8192

Table 5.5: GPU Test 2: cuFFTXt on one node with two GPU

Dims	Forward [ms]	Backward [ms]	Total [ms]	Mem/Proc [MB]
256	3	3	539	128
512	31	31	858	1024
1024	264	264	5370	8192

one GPU. The overhead of coordinating between the two GPUs might be the reason for this anomaly, which overshadowed the benefits. This might have occurred due to several reasons, such as the use of high-latency or low-bandwidth network and large data transfer. Because the multi-node GPU implementation with cuFFTMp was not run and tested, the direct multi-node data is currently unavailable. However, it can be inferred, from the single-node multi-GPU behavior, that simply adding more GPUs without addressing communication would not help much. cuFFTMp is explicitly designed to alleviate this by using RDMA and GPU-driven transfers, and it can be anticipated that on a system with multiple nodes (`node3/node4`) of A100 GPUs (which have very fast NVLink/NVSwitch and PCIe4/5), cuFFTMp would show much better scaling.

The Figures 5.5 and 5.6 illustrate the advantage of GPU computing for FFT workloads, as well as the challenges of multi-GPU scaling. The single GPU absolutely excels: for 1024^3, achieving 11× speedup over 32 CPU cores, where one P100 GPU did in 2.24 seconds what an entire node of CPUs did in 25.5 seconds. This kind of speedup is in line with the differences in memory bandwidth and flop/s between one P100 and one CPU socket. It demonstrates that for FFT-heavy calculations, modern GPUs can significantly accelerate computation.

5.3 Comparative CPU-GPU-Multi-GPU Evaluation

For relatively smaller problems (like $N = 256^3$ or $N = 512^3$), a single GPU clearly outperforms a multi-core CPU. For larger problems ($N = 1024^3$ and beyond), this gap widens further. At $N = 1024^3$, one GPU vs. 32 CPU cores gave around 11× speedup and around 78× vs 1 core. Hypothetically, the GPU would still be about 6x faster, even if the CPU cores had scaled perfectly to 64 cores. These results highlight how GPU acceleration becomes more attractive as problem size grows: the GPU's fixed

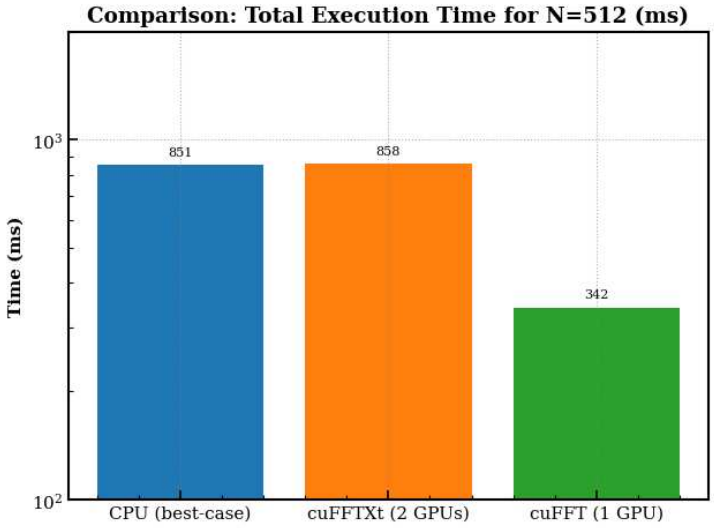

Figure 5.5: Total execution time for a 512^3 grid using (i) CPU (MPI) with 64 processes – CPU best-case, (ii) 2× GPUs (single-node, using cuFFTXt), and (iii) 1× GPU (single GPU).

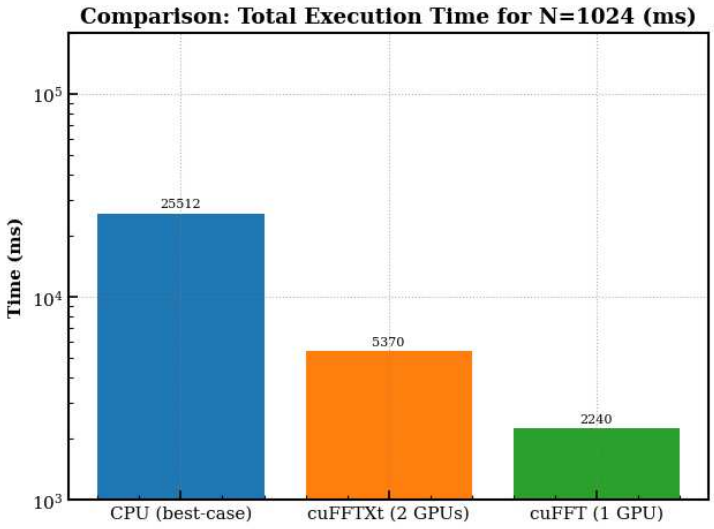

Figure 5.6: Total execution time for a 1024^3 grid for (i) CPU best-case (32 MPI processes gave the best time), (ii) 2× GPUs (cuFFTXt), and (iii) 1× GPU.

overhead (memory transfer, kernel launch) becomes relatively small compared to the large computation, and it can fully utilize its parallel hardware, whereas the CPU network overhead grows.

On the other hand, the multi-GPU (2-GPU) results, while not showing speedup over 1 GPU, still showed a benefit over CPUs. For 1024^3, 2 GPUs (5.37 s) were about 4.7× faster than 32 CPU cores (25.5 s). So if one had the choice between using 32 CPU cores or 2 GPUs for that problem, the 2 GPUs would be clearly better. However, resorting to a single GPU would be more practical for such cases because the under-performance of double-GPU compared to that of the single-GPU implementation can significantly be noted. The communication bottlenecks provide a reliable comparative insight: the CPU solver's bottleneck was MPI All-to-All; the multi-GPU solver's bottleneck was GPU-GPU data exchange, both essentially being the same kind of operation (transpose). On CPUs, adding more ranks increased the overhead; on GPUs, adding a second GPU without high-speed interconnect also incurred overhead.

Chapter 6

Conclusion and Future Work

The original objective of this project was to integrate multi-node, multi-GPU acceleration into Quantum ESPRESSO's FFTXlib by utilizing NVIDIA's cuFFTMp library. Given the intricate complexities and deeper inter-dependencies within the QE codebase, a direct integration into FFTXlib was difficult to achieve in a limited time frame, especially in the scope of this project. Hence, the focus was strategically shifted to a simplified, yet representative subproblem within Quantum ESPRESSO, the Poisson Solver, which enabled an independent and controlled study of distributed multi-GPU FFT performance. The primary goal of this redirection was to validate whether cuFFTMp-based multi-node GPU acceleration could yield tangible speedups for FFT-dominated routines and provide insights on the feasibility of extrapolation to the QE's FFTXlib.

A multi-GPU Poisson solver was implemented using the CUDA Programming paradigms and validated to study the effectiveness of multi-GPU scaling. Performance evaluations demonstrated significant acceleration on GPUs; for instance, a single NVIDIA P100 GPU surpassed highly parallel CPU implementations by nearly an order of magnitude for large 3D grids. These results emphasize the potential of offloading FFT-intensive workloads to GPUs to substantially reduce simulation runtimes. The conduction of strong scaling tests further illuminated both the strengths and bottlenecks of multi-GPU systems. While CPU-based approaches saturated early due to communication overheads, GPU-based implementations revealed limitations primarily stemming from inter-GPU communication, and thus present opportunities for optimization through communication-overlap techniques (e.g., using NVSHMEM, asynchronous data exchanges). Overall, the GPU implementation proved the feasibility and benefit of offloading FFT-based solvers to accelerators. The single-GPU case achieved excellent performance. The multi-GPU case, while not yielding speedup in our tests, provided valuable insight into what needs to be improved: better GPU interconnect utilization and possibly algorithmic changes to overlap computation/communication. This is precisely where a library like cuFFTMp is supposed to help.

Based on the study insights and outcomes, future work prospects in the regard of this project can be:

1. **Extended Multi-GPU and Multi-Node Benchmarking:** Comprehensive testing on advanced nodes such as those with A100 GPUs interconnected via NVLink/NVSwitch (`node3/node4`) needs to be further done, with a particular focus on scaling experiments across two or more nodes to evaluate cuFFTMp's internode performance, leveraging high-bandwidth InfiniBand networks.

2. **Integration into Quantum ESPRESSO's FFTXlib:** After completely validating the tests on the subproblem, the next logical step is to progressively integrate cuFFTMp into FFTXlib. A practical pathway would involve selectively replacing the existing MPI-based 3D FFTs with cuFFTMp-enabled routines, initially targeting critical kernels in the relevant modules. This staged approach would ensure functional correctness and performance gains without destabilizing the broader QE codebase.

3. **Optimization of Communication Overlap:** Utilizing CUDA streams to asynchronously overlap data transfers and computation, tuning NVSHMEM parameters, and experimenting with double-buffering strategies could push the GPU scaling further, especially in the multi-node regime.

Appendices

A Quantum ESPRESSO Repository Architecture

QE is an integrated suite composed of several packages/modules, each designed for specific tasks in computational materials science, such as PWscf, CP, PHonon, NEB and many more. The source code is organized into a number of directories, each containing specific modules and files for different functionalities. All these components share common libraries and modules within QE, such as Modules/ (common routines for I/O, symmetry operations, etc.), FFTXlib/ (for performing FFTs as discussed in this report), and LAXlib/ (linear algebra routines). A high-level overview of the repository architecture is shown in Figure A, highlighting some of the dependencies uncovered between different modules during the course of this study. The GPU implementation of 3D Fast Fourier Transform operations in Quantum ESPRESSO's FFTXlib is distributed across several source files in the `src` directory. The scatter and scaling routines are implemented `fft_scatter_2d_gpu.f90` and `fft_scalar_cuda.f90` respectively, with these files containing the primary CUDA implementation that interfaces with the cuFFT library. The implementation is structured around key routines including `cft3d_gpu` and `cft3ds_gpu` for complex-to-complex transforms, `tg_cft3s_gpu` for task-group parallel operations, and dedicated plan management routines such as `create_plan_gpu` and `destroy_plan_gpu`. The dependency chain flows from high-level QE modules through `fft_interfaces.f90` and `fft_parallel.f90` to low-level modules like `fft_scalar_cuda.f90`, which ultimately interface with the cuFFT library. Additional dependencies include DeviceXlib for GPU memory management, the CUDA runtime for device operations, and MPI for multi-GPU configurations.

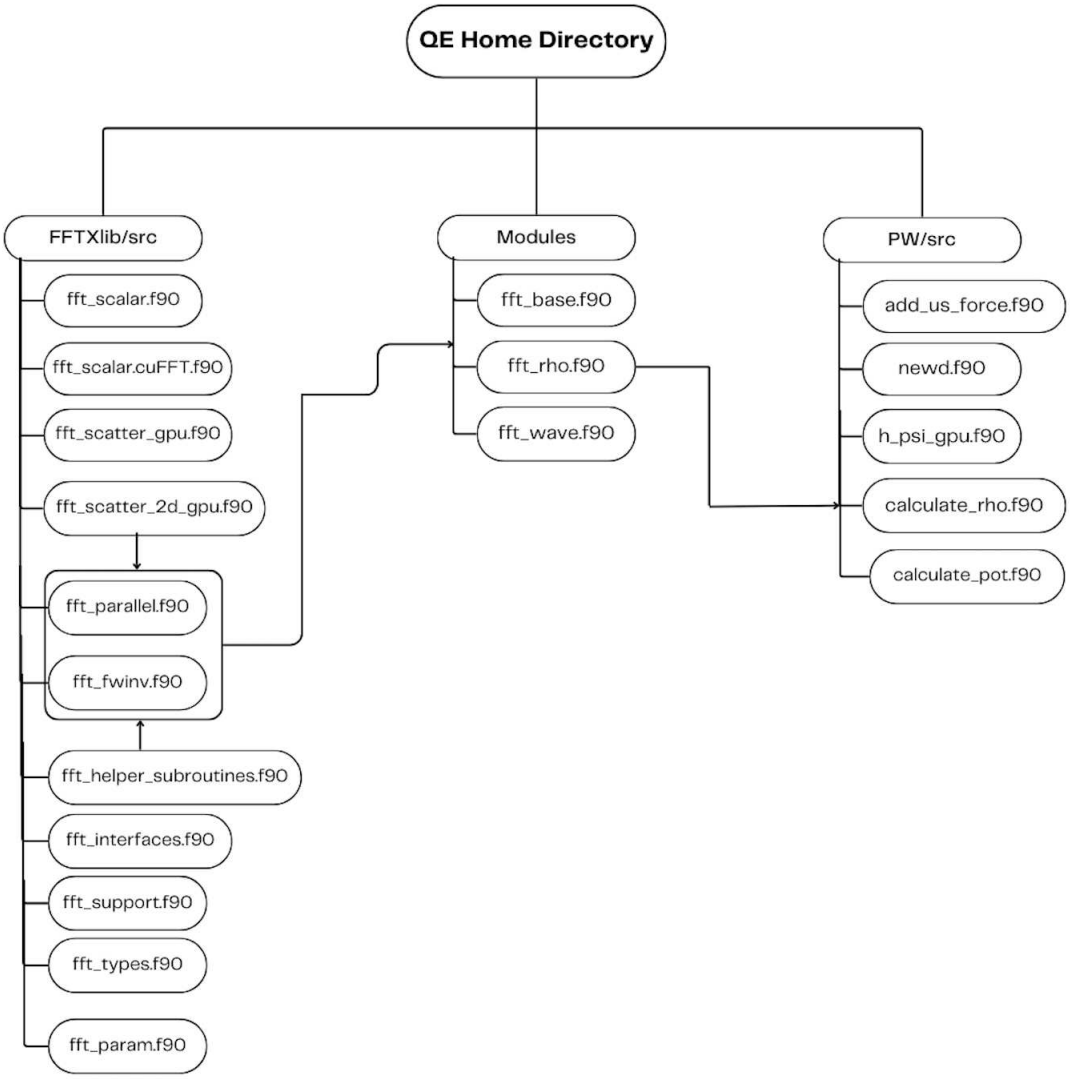

Figure A: Hierarchical Architecture of the Quantum ESPRESSO's Official Repository depicting some of the files implementing GPU-enabled routines along with their dependencies

References

[1] Fabrizio Ferrari Ruffino, Laura Bellentani, Giacomo Rossi, Fabio Affinito, Stefano Baroni, Oscar Baseggio, Pietro Delugas, Paolo Giannozzi, Jakub Kurzak, Ye Luo, Ossian O'Reilly, Sergio Orlandini and Ivan Carnimeo. Quantum ESPRESSO towards performance portability: GPU offload with OpenMP. *Procedia Computer Science*, 240:52–60, 2024. Proceedings of the First EuroHPC user day.

[2] Pietro Delugas, Ivan Carnimeo. Overview of QUANTUM ESPRESSO suite of codes and main features. ENCCS-MaX School 2024.

[3] Filippo SPIGA. GPU Direct Approach for Parallel 3D-FFT in Quantum ESPRESSO. University of Cambridge.

[4] Ivan Carnimeo, Fabio Affinito, Stefano Baroni, Oscar Baseggio, Laura Bellentani, Riccardo Bertossa, Pietro Davide Delugas, Fabrizio Ferrari Ruffino, Sergio Orlandini, Filippo Spiga, and Paolo Giannozzi. Quantum ESPRESSO: One Further Step toward the Exascale. *Journal of Chemical Theory and Computation*, 19(20):6992–7006, 2023.

[5] Phillips E. Ruetsch G. Fatica M. Spiga F. Giannozzi P. Romero, J. A Performance Study of Quantum ESPRESSO's PWscf Code on Multi-core and GPU Systems. *Springer, Cham.*, 2017.

[6] Paolo Giannozzi, Oscar Baseggio, Pietro Bonfà, Davide Brunato, Roberto Car, Ivan Carnimeo, Carlo Cavazzoni, Stefano de Gironcoli, Pietro Delugas, Fabrizio Ferrari Ruffino, Andrea Ferretti, Nicola Marzari, Iurii Timrov, Andrea Urru, and Stefano Baroni. Quantum ESPRESSO toward the exascale. *The Journal of Chemical Physics*, 152(15):154105, 04 2020.

[7] Pietro Bonfà. Quantum ESPRESSO on HPC and GPU systems: parallelization and hybrid architectures. MaX School on Advanced Materials and Molecular Modelling with Quantum ESPRESSO 2021.

[8] *User's Guide for Quantum ESPRESSO (v.7.3.1).*

[9] Xuejun Gong and Andrea Dal Corso. An alternative GPU acceleration for a pseudopotential plane-waves density functional theory code with applications to metallic systems. *Computer Physics Communications*, 308:109439, 2025.

[10] Yifeng Chen, Xiang Cui, and Hong Mei. Large-scale FFT on GPU clusters. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, page 315–324. Association for Computing Machinery, 2010.

[11] Akira Nukada, Kento Sato, and Satoshi Matsuoka. Scalable multi-GPU 3-D FFT for TSUBAME 2.0 Supercomputer. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–10, 2012.

[12] Jose L. Jodra, Ibai Gurrutxaga, Javier Muguerza, and Ainhoa Yera. Solving Poisson's equation using FFT in a GPU cluster. *Journal of Parallel and Distributed Computing*, 102:28–36, 2017.

[13] Michael Wagner, Victor López, Julián Morillo, Carlo Cavazzoni, Fabio Affinito, Judit Giménez, and Jesús Labarta. Performance Analysis and Optimization of the FFTXlib on the Intel Knights Landing Architecture. In *2017 46th International Conference on Parallel Processing Workshops (ICPPW)*, pages 243–250, 2017.

[14] Stefano de Gironcoli. New FFT data distribution to improve scalability. Scuola Internazionale Superiore di Studi Avanzati Trieste-Italy.

[15] Orlando Ayala and Lian-Ping Wang. Parallel implementation and scalability analysis of 3D Fast Fourier Transform using 2D domain decomposition. *Parallel Computing*, 39(1):58–77, 2013.

[16] Alan Ayala, Stan Tomov, Miroslav Stoyanov, Azzam Haidar, and Jack Dongarra. Accelerating Multi - Process Communication for Parallel 3-D FFT. In *2021 Workshop on Exascale MPI (ExaMPI)*, pages 46–53, 2021.

[17] Thomas Koopman and Rob H. Bisseling. Minimizing Communication in the Multidimensional FFT. *SIAM Journal on Scientific Computing*, 45(6):C330–C347, 2023.

[18] Naga K. Govindaraju, Brandon Lloyd, Yuri Dotsenko, Burton Smith, and John Manferdelli. High performance discrete Fourier transforms on graphics processors. In *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, pages 1–12, 2008.

[19] Foster, Ian T. and Worley, Patrick H. Parallel Algorithms for the Spectral Transform Method. *SIAM Journal on Scientific Computing*, 18(3):806–837, 1997.

[20] Paolo Giannozzi. *Lecture notes on Numerical Methods in Quantum Mechanics*. University of Udine, 2021.

[21] Gavin J. Pringle, Joahcim Hein. Parallel Fast Fourier Transforms. EPCC.

[22] Dr. Niranjan.S.Ghaisas. Lecture Notes on Introduction to Parallel Scientific Computing. IIT Hyderabad.

[23] CUDA v12.3 User Documentation, cuFFT v12.3 User Documentation, cuFFTMp v11.2.3 User Documentation. NVIDIA Corporation. 2024.

[24] Steven L. Brunton and J. Nathan Kutz. *Data Driven Science and Engineering*. University of Washington, 2017.

[25] Jason Sanders and Edward Kandrot. *CUDA by Example*. NVIDIA Corporation, 2011.

[26] David B.Kirk and Wen-mei W.Hwu. *Programming Massively Parallel Processors. 3rd Edition*. Morgan Kaufmann, 2017.