MASTER IN HIGH PERFORMANCE COMPUTING

Strategies for GPU-Optimized Small Size Jobs in QUANTUM ESPRESSO Combining OpenMP and OpenACC

Supervisor:
Pietro DELUGAS
Stefano DE GIRONCOLI

Candidate:
Nisha

11th EDITION
2024–2025

Master in
High Performance Computing

Contents

Contents iii

1 Preface 1

2 Theoretical Foundations – DFT and Quantum ESPRESSO 5
2.1 From Many-Body Problem to Kohn-Sham Equations 5
2.2 QUANTUM ESPRESSO Package 6
2.2.1 Structure of the PW Code 7
2.2.2 Iterative Diagonalization Methods 8
2.2.3 Davidson Diagonalization 9

3 Diagonalization and Computational Methods 11
3.1 Bloch's Theorem and k-Point Sampling 11
3.1.1 Bloch's Theorem 11
3.1.2 k-Point Sampling in Density Functional Theory 12
3.1.3 Computational Independence of k-points 12
3.2 The Kohn-Sham Eigenvalue Problem 13
3.2.1 From Kohn-Sham Equations to Matrix Form 13
3.2.2 Structure of Hamiltonian Matrix 14
3.2.3 The Overlap Matrix 15
3.3 Computational Implementation: Pseudopotentials, FFTs, and Dense Linear Algebra 16
3.3.1 Pseudopotentials: Reducing Computational Complexity 16
3.3.2 Fast Fourier Transforms: Applying the Local Potential 17
3.3.3 BLAS for Linear Algebra 18
3.4 Computational Cost Analysis 19
3.4.1 Cost Distribution in Davidson Iterations 20
3.4.2 Parallelization Opportunities 20
3.5 Summary and Motivation 21

4 CPU Parallelization via k-Point Batching 23
4.1 Motivation and Strategy 23
4.2 The cb_davidson Miniapp 24
4.2.1 Rationale for Miniapp Development 24
4.2.2 Miniapp Structure 25
4.3 OpenMP Implementation of k-Point Batching 25
4.3.1 Parallel Region Design 25
4.3.2 Data Management and Scope 27
4.3.3 Runtime Thread Control 28

4.4 Thread Safety Challenge: The Timing System . . . 28
4.4.1 Initial Problem Manifestation . . . 28
4.4.2 Root Cause: Non-Thread-Safe Clocks . . . 29
4.4.3 Clock Context Mismatch . . . 29
4.4.4 Solution: Per-Thread Timing Architecture . . . 30
4.4.5 Consistent Clock Placement . . . 32
4.5 Performance Results . . . 33
4.5.1 Execution Time and Speedup . . . 33
4.5.2 Scaling Analysis . . . 34
4.6 Summary and Motivation . . . 34

5 GPU Asynchronous Stream Execution **37**
5.1 The GPU Underutilization Problem . . . 37
5.2 OpenACC as the GPU Programming Model . . . 38
5.2.1 OpenACC Directives . . . 38
5.2.2 Central role of Async Clause . . . 38
5.3 CUDA Streams and the Asynchronous Execution Model . . . 39
5.3.1 Concept of a CUDA Stream . . . 39
5.3.2 Relationship Between OpenACC Queues and CUDA Streams . . . 39
5.3.3 Thread Private Stream Identification . . . 40
5.3.4 Asynchronous Memory Transfers . . . 40
5.4 Synchronization Strategy . . . 40
5.4.1 Ordering Within a Stream . . . 40
5.4.2 Synchronization Across Streams . . . 41
5.4.3 Host and Device Consistency for Convergence Checks . . . 41
5.4.4 MPI Communication and Device Pointers . . . 42
5.5 Management of GPU Library Handles . . . 42
5.5.1 cuBLAS Handle Management . . . 42
5.5.2 cuSOLVER Handle via laxlib . . . 43
5.5.3 FFT Operations and the fftw_locker . . . 44
5.6 Profiling and GPU utilization Results . . . 44
5.6.1 Wall Time and Speedup . . . 45
5.6.2 Timeline Analysis . . . 45
5.6.3 The FFT Lock Bottleneck . . . 47
5.6.4 Kernel Distribution Analysis . . . 47
5.6.5 Discussion . . . 48
5.7 Summary . . . 48

6 Conclusions **51**
6.1 Summary of Contributions . . . 51
6.2 Future Developments . . . 52
6.3 Closing Remarks . . . 53

Bibliography **55**

1 Preface

This thesis addresses the problem of GPU underutilization in scientific computing applications, specifically in the context of plane-wave density functional theory calculations performed with Quantum ESPRESSO.

Quantum ESPRESSO (QE) is a widely used open-source software suite for electronic-structure calculations and materials simulations from first principles. Based on the plane-wave pseudopotential implementation of Density Functional Theory (DFT), it enables prediction of electronic structure, total energies, forces, and stresses without relying on empirical parameters, and has become an essential tool across condensed matter physics, materials science, and computational chemistry.

The core computational workflow in Quantum ESPRESSO's PWscf (Plane Wave Self-Consistent Field) module centres on the iterative solution of the Kohn-Sham equations to determine electronic wavefunctions and energies. This procedure requires repeated diagonalization of the Hamiltonian matrix at each k point in the Brillouin zone. Among the most computationally demanding stages is the Davidson iterative diagonalization algorithm, which typically consumes about 60 to 80 percent of the total runtime in standard self-consistent field calculations. Because the algorithm must be applied at every k point during each self consistent iteration, and each diagonalization itself involves multiple internal iterations to reach the desired accuracy, even small performance improvements in this routine can translate into substantial reductions in overall simulation time.

The GPU Underutilization Problem in Small Jobs

Over the last decade, high-performance computing (HPC) has transitioned from traditional CPU-only architectures toward heterogeneous systems that pair CPUs with Graphics Processing Units (GPUs). GPU-accelerated systems are now widely used in HPC, as they support many-core processors with high flop rates and memory bandwidth, as well as energy efficiency and a mature software ecosystem. Quantum ESPRESSO has been progressively ported to GPU architectures using OpenACC directives and CUDA libraries, allowing the dominant operations, such as FFT-based potential applications and dense linear algebra to execute on the GPU.

However, a fundamental mismatch arises for small-sized jobs, namely calculations involving systems with few bands and low plane-wave cutoffs. In these cases, although the calculation may involve many k-points, each individual Davidson solution at a single k-point provides insufficient computational work to saturate a modern GPU. The device executes a short burst of computation, then sit idles while the host prepares the next operation. This underutilization is not a fault of the GPU porting itself; it is a structural consequence of processing k-points sequentially when each individual solution is too

small to keep the device busy.

Thesis Scope and Objectives:
This thesis develops and demonstrates two complementary strategies for addressing GPU underutilization in small-sized Quantum ESPRESSO jobs, combining OpenMP threading and OpenACC GPU directives in a two-level parallelization framework.

Strategy 1 — CPU-level k-point batching via OpenMP (Chapter 4): The mathematical independence of the Kohn-Sham eigenvalue problem at different k-points enables embarrassingly parallel execution. Multiple k-points are processed concurrently by OpenMP threads on the host CPU, each thread managing its own independent Davidson solve. This exposes coarse-grained parallelism that complements the fine-grained parallelism within each solve and provides the structural foundation for GPU concurrency.

Strategy 2 — GPU-level concurrent stream execution via OpenACC (Chapter 5): Building on the OpenMP batching framework, each concurrent k-point is assigned its own CUDA stream obtained through the OpenACC asynchronous queue mechanism. All GPU operations for a given k-point, including kernel launches, memory copies, cuBLAS matrix multiplications, and cuSOLVER diagonalizations, are enqueued to that stream without blocking the host thread. Multiple streams execute concurrently on the GPU, filling idle gaps that a single sequential stream would leave. Per-batch library handles (cuBLAS, cuSOLVER) and thread-private stream identity variables ensure complete isolation between concurrent k-points without requiring serialization locks.

To develop and validate these strategies in a controlled environment isolated from the complexity of the full QE codebase ($\approx$ 500,000 lines of Fortran), a standalone miniapp (`cb_davidson`) was created by extracting the Davidson solver and its essential supporting routines from the QE library infrastructure. This miniapp approach is a deliberate methodological decision because it eliminates coupling effects that would arise from altering the entire PWscf code, enables rapid prototyping, facilitates clear benchmarking, and allows direct attribution of performance differences to specific implementation choices. Since the miniapp uses the same Davidson solver (cegterg) and library stack (LAXlib, FFTXlib, and UtilXlib) as production Quantum ESPRESSO, the techniques shown there are immediately transferable to the complete code.

A primary motivation for using the miniapp approach is the substantial complexity of integrating the batching strategy into the full Quantum ESPRESSO codebase. The production **h_psi** routine relies on globally shared workspaces without a batch dimension, assumes single–k-point execution, and incorporates numerous additional physical contributions. Making it thread-safe for concurrent k-point processing would therefore require extensive refactoring across many interdependent arrays and source files, which lies beyond the scope of this thesis.

Thesis Organization

Chapter 2 introduces the theoretical foundations of Density Functional Theory. The many-body Schrödinger equation is presented, followed by the Born-Oppenheimer approximation, the Hohenberg-Kohn theorems, and the Kohn-Sham formulation that reduces the interacting many-body problem to a self-consistent set of single-particle equations. The structure of Quantum ESPRESSO's PWscf code is described, including the SCF iteration workflow and the role of iterative diagonalization in solving the Kohn-Sham eigenvalue problem.

Chapter 3 develops the computational framework underlying the performance challenges addressed in this thesis. Bloch's theorem establishes k-point sampling as the source of the embarrassingly parallel structure exploited by Strategy 1. The plane-wave formulation of the Kohn-Sham matrix is derived, and the computational cost of each Davidson phase is analyzed to identify the dominant operations, such as FFT-based potential applications and BLAS matrix multiplications, which motivate GPU offloading. The chapter concludes with a quantitative analysis of why sequential k-point execution underutilizes GPU hardware for small jobs, directly motivating the two strategies developed in Chapters 4 and 5.

Chapter 4 presents Strategy 1: CPU parallelization through OpenMP k-point batching. The cb_davidson miniapp is introduced, and its design rationale is explained. The two-level loop structure and batched array layout are described in detail. A significant technical challenge, that is race conditions in the timing infrastructure encountered during implementation, which is caused by shared module-level arrays, is diagnosed and resolved through a thread-private clock indexing architecture. Performance results quantify the speedup achieved on multi-core CPU nodes.

Chapter 5 presents Strategy 2: GPU acceleration via asynchronous stream-based execution. The OpenACC programming model is described, with particular attention to the async clause and its mapping to CUDA streams through acc_get_cuda_stream. The multi-library handle architecture, which includes per-batch cuBLAS handles, pre-allocated cuSOLVER handles via LAXlib, and batched cuFFT workspaces, is explained as the mechanism that eliminates inter-k-point serialization. Synchronization strategy at three levels (intra-stream ordering, convergence testing, and the batch boundary barrier) is analyzed. Profiling results on Leonardo's A100 GPUs demonstrate the improvement in GPU utilization achieved by the combined OpenMP and OpenACC strategy.

Chapter 6 presents the conclusions of this thesis. The contributions of both strategies are summarized, with emphasis on the measured performance improvements and the bottlenecks identified through profiling. Future development directions are outlined, including the replacement of cuSOLVER with a batched small-matrix eigensolver, and the integration of the validated strategies into the full Quantum ESPRESSO production codebase.

2 Theoretical Foundations – DFT and Quantum ESPRESSO

2.1 From Many-Body Problem to Kohn-Sham Equations

Understanding the electronic structure of materials requires solving the many-body Schrödinger equation,

$$\hat{H}\psi_i(\{\vec{r}\}, \{\vec{R}\}) = E_i\psi_i(\{\vec{r}\}, \{\vec{R}\}), \tag{2.1}$$

where $\hat{H}$ contains the kinetic energies of N electrons and M nuclei together with electron—electron, electron–nuclear, and nucleus–nucleus Coulomb interactions. In atomic units, the Hamiltonian includes non-separable terms such as $|r_i - r_j|^{-1}$, which makes the full $3N + 3M$ coordinate problem impossible to solve analytically or numerically for any realistic solid-state system. An approximation is therefore unavoidable.

The Born–Oppenheimer approximation (BOA) [BO27] provides the first major simplification by exploiting the large mass ratio between nuclei and electrons. Since nuclear positions move and fluctuate much more slowly, they may be treated as fixed parameters in the electronic problem, eliminating nuclear kinetic contributions and reducing the total Hamiltonian to an electronic one for fixed $\{\vec{R}_A\}$. While this separation simplifies the problem, solving the electron–electron-interacting system remains computationally prohibitive even for modest numbers of electrons.

Density Functional Theory (DFT) provides a practical route to treat many-electron systems by replacing the wavefunction $\psi(\vec{r}_1, \ldots, \vec{r}_N)$ with the ground-state electron density $n(\vec{r})$ [Mar20]. The Hohenberg-Kohn theorems [HK64] establish (i) that the ground-state density uniquely defines the external potential, and (ii) that the true ground-state density minimizes the energy functional $E[n]$. Kohn and Sham [KS65] reformulated the theory by introducing a system of non-interacting electrons that reproduces the exact ground-state density, leading to the Kohn-Sham equations:

$$\left[-\frac{1}{2}\nabla^2 + V_{\text{eff}}(\vec{r})\right]\phi_i^{\text{KS}}(\vec{r}) = \epsilon_i\phi_i^{\text{KS}}(\vec{r}), \tag{2.2}$$

where

$$V_{\text{eff}}(\vec{r}) = V_{\text{ext}}(\vec{r}) + U_{\text{coul}}\vec{r} + V_{\text{XC}}\vec{r} \tag{2.3}$$

contains the external (nuclear) field, classical Hartree term, and the exchange–correlation (XC) potential. The XC potential captures all non-trivial many-body effects of the interacting electron system, but its exact form is unknown and must be approximated in practice. Several approximations exist ranging from local density approximation (LDA), to the generalized gradient approximation (GGA), and more other variants such

as meta-GGA and hybrid functionals. The electron density is obtained self-consistently from

$$n(\vec{r}) = \sum_{i=1}^{N} |\phi_i^{\mathrm{KS}}(\vec{r})|^2. \tag{2.4}$$

Solving the Kohn-Sham equations for each k-point at every self-consistent-field (SCF) iteration forms the computational core of modern first-principles electronic-structure calculations and constitutes the numerical workload studied in later chapters.

Plane-Wave Basis and Pseudopotentials

In practical solid-state DFT implementations, the plane-wave basis combined with pseudopotentials has become one of the most efficient and widely used formulations, following developments summarized, for example, in [Pic89]. Plane waves form a complete, orthonormal basis naturally suited to periodic boundary conditions and to reciprocal-space formulations of the kinetic operator. The Kohn-Sham orbitals are expanded as:

$$\phi_{j\vec{k}}(\vec{r}) = \sum_{\vec{G}} C_j(\vec{k} + \vec{G})\, e^{i(\vec{k}+\vec{G})\cdot\vec{r}}, \tag{2.5}$$

where $\vec{G}$ are reciprocal-lattice vectors. The kinetic-energy cutoff determines the number of included plane waves and thus the computational cost. However, core electrons oscillate rapidly near the nuclei and would require extremely high cutoffs. To avoid this, pseudopotential methods replace the all-electron potential with a smoother effective potential in which valence electrons feel a modified ionic field, while still reproducing essential scattering properties. Norm-conserving, ultrasoft, and projector-augmented-wave (PAW) pseudopotentials are standard choices. This approach drastically reduces the number of plane waves required, enabling the scalable and efficient electronic-structure calculations used throughout this thesis.

Together, the plane-wave basis and pseudopotential methods form the computational foundation for the Davidson eigensolver, k-point parallelization strategy, and GPU-accelerated algorithms discussed in Chapters 4 and 5.

2.2 Quantum ESPRESSO Package

Quantum ESPRESSO is an open-source, modular distribution of software tools designed for electronic-structure calculations and materials modelling within density-functional theory (DFT) and plane-wave basis sets, and pseudopotential frameworks described above [Gia+17]. Its development philosophy emphasizes interoperability, extensibility, and high performance: rather than a single monolithic code, it consists of coordinated core components and community-contributed modules, all relying on shared file formats and optimized numerical libraries [Gia+20]. The project aims both to support innovation in electronic-structure methodologies and to provide a robust, efficient environment accessible to a broad user community and scalable to modern, massively parallel architectures.

In practical terms, QE offers a wide range of capabilities for chemically realistic simulations of materials. It solves the Kohn-Sham equations using periodic boundary conditions, enabling the description of crystals, liquids, amorphous solids, and finite systems via supercells. Many types of pseudopotentials are supported, including norm-conserving, ultrasoft, and projector-augmented-wave (PAW) datasets, together with a large variety of exchange–correlation functionals such as LDA, GGA, DFT+U, meta-GGA, and several hybrid functionals. Its main simulation features include ground-state electronic-structure calculations, full structural optimizations of atoms and cell parameters, spin-polarized and non-collinear magnetism with or without spin–orbit coupling, and *ab initio* molecular dynamics using either Born–Oppenheimer or Car–Parrinello formulations.

The suite also integrates advanced methods such as density-functional perturbation theory for phonons, dielectric and vibrational properties, electron–phonon and phonon–phonon interactions; transition-state and reaction-path searches via the nudged elastic band (NEB) method. A set of post-processing tools allows the extraction and visualization of densities, potentials, densities of states, STM images, and other derived quantities, while interfaces exist for external software relying on DFT inputs. Through this combination of modularity, methodological breadth, and computational efficiency, QE provides a comprehensive, flexible platform for first-principles investigations of materials from the nanoscale to extended condensed-phase systems.

2.2.1 Structure of the PW Code

PW is the core module of Quantum ESPRESSO for ground state electronic structure calculations. It implements the Kohn-Sham density functional theory framework described in Section 2.3 and solves the self-consistent equations for both molecular and periodic systems. Beyond basic electronic structure calculations, PW includes functionality for:

- Structural relaxation using quasi-Newton methods Broyden–Fletcher–Goldfarb–Shanno (BFGS) algorithm.
- Variable-cell optimization for crystals.
- Force and stress calculations.
- Band structure and density of states analysis.

For this thesis, we focus on the self-consistent solution of Kohn-Sham equations for fixed atomic geometries, with particular emphasis on the iterative diagonalization step. Figure 2.1 illustrates the hierarchical structure of a typical PW calculation. The outermost loop handles structural optimization, while the inner self-consistent field (SCF) loop iterates over the electron density until convergence. The Davidson method and other iterative eigensolvers are used for diagonalization. Every k-point undergoes this diagonalization. The number of electrons in the unit cell determines the number of occupied KS orbitals.

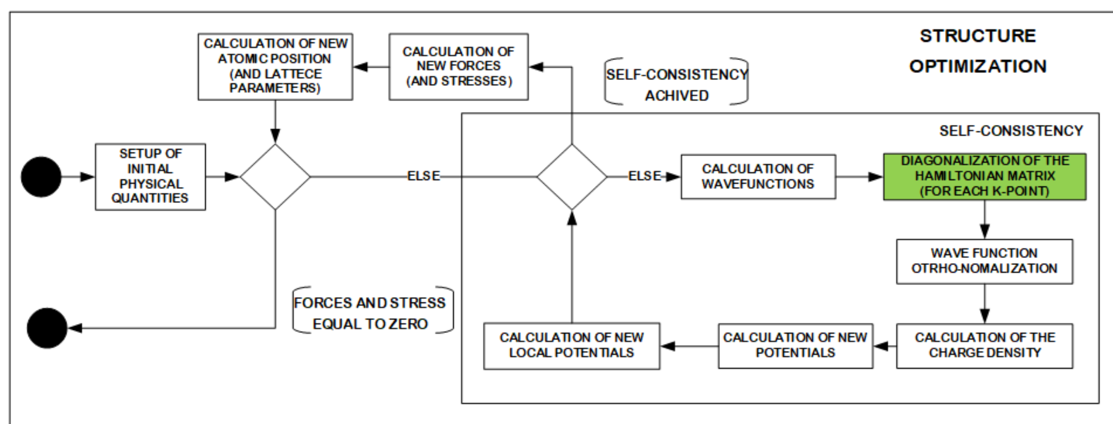

Figure 2.1: Schematic view of the PWscf internal steps.

An important performance issue arises from the inherent structure of the PWscf computational workflow. The diagonalization stage within the k-point loop must be performed for each k point during every self-consistent field iteration. In systems of limited size, characterized by few bands and low plane-wave cutoffs, each individual diagonalization step provides insufficient computational work to fully engage a modern GPU, despite the calculation involving many k-points. On GPU-accelerated platforms, this introduces a fundamental inefficiency: the device completes one k-point solution and then remains idle while the host prepares the next task, leading to systematic underutilization of available hardware resources. This operating regime of small problem sizes, together with the methods required to mitigate the resulting GPU underutilization, constitutes the primary focus of this thesis.

2.2.2 Iterative Diagonalization Methods

The combination of reduced computational cost, favorable memory scaling, and matrix-free operation makes iterative methods indispensable for modern electronic-structure calculations. Iterative diagonalization provides an efficient alternative to dense diagonalization, which would be impractical for plane-wave basis sets of such large dimensionality, and it reduces both the computational cost and the memory requirements to operations that scale linearly with the size of the plane-wave basis. Based on the needs and the resources, the user can choose the algorithm to solve the following equations:

$$\hat{H}\psi_i = \epsilon_i \hat{S}\psi_i, \quad \text{where} \quad i = 1, 2, \ldots, N \tag{2.6}$$

Here, N denotes the number of occupied states, and $\hat{S}$ is the overlap operator. The eigenvectors are normalized to satisfy the generalized orthonormality constraints $\langle\psi_i|S|\psi_j\rangle = \delta_{ij}$. The block Davidson method is computationally efficient with respect to the number of Hamiltonian applications $H|\psi\rangle$ it requires, although this efficiency comes at the cost of high memory usage. Quantum ESPRESSO incorporates also other

multiple iterative eigensolvers, each characterized by different computational properties and performance tradeoffs.

2.2.3 Davidson Diagonalization

The Davidson algorithm is the default and most efficient method for most systems encountered in materials science applications [Dav75]. The procedure is iterative: during each cycle, the Hamiltonian operator is applied to trial wavefunctions, residual vectors are evaluated, and the subspace is enlarged through the addition of preconditioned residuals. The Davidson method begins by choosing an initial set of orthonormal trial orbitals $\psi_i^{(0)}$, along with the corresponding trial eigenvalues defined as:

$$\varepsilon_i^{(0)} = \langle \psi_i^{(0)} | \hat{H} | \psi_i^{(0)} \rangle \tag{2.7}$$

The quality of the approximate eigenpair $(\varepsilon_i^{(0)}, \psi_i^{(0)})$ is then evaluated through the residual (correction) vector, which measures the deviation from an exact eigenvector of the Hamiltonian:

$$r_i = \hat{H}\psi_i^{(0)} - \varepsilon_i^{(0)} \hat{S}\psi_i^{(0)}. \tag{2.8}$$

The eigenvalue problem is then projected onto a 2N-dimensional subspace spanned by a reduced basis ϕ^0, where the first N basis functions are the trial orbitals, and the remaining N functions are the corresponding correction vectors.
Within this subspace, the generalized eigenvalue problem:

$$\sum_{k=1}^{2N} (H_{jk} - \varepsilon_i S_{jk}) c_k^{(i)} = 0 \tag{2.9}$$

is solved, where $H_{jk} = \langle \phi_j^{(0)} | \hat{H} | \phi_k^{(0)} \rangle$ and $S_{jk} = \langle \phi_j^{(0)} | \hat{S} | \phi_k^{(0)} \rangle$. This procedure yields an updated set of trial eigenvectors and eigenvalues.

$$\phi_i^{(1)} = \sum_{j=1}^{2N} c_j^{(i)} \phi_j^{(0)}, \qquad \varepsilon_i^{(1)} = \langle \phi_i^{(1)} | \hat{H} | \phi_i^{(1)} \rangle \tag{2.10}$$

The process is repeated iteratively until the desired level of convergence is reached. In the PWscf implementation, the Davidson algorithm is primarily managed by the routines regterg and cegterg, which perform real and complex iterative generalized eigenvalue calculations, respectively. The routines `diaghg` and `cdiaghg` handle real and complex generalized diagonalization of the Hamiltonian, while `h_psi`, `s_psi`, and `g_psi` are code-specific procedures used within this framework.
In addition to Davidson, QUANTUM ESPRESSO includes several alternative iterative eigensolvers designed for specific computational scenarios [Dai+14] [Dai+15].
The projected preconditioned conjugate gradient method reduces memory requirements by updating bands in small groups rather than maintaining a large subspace. While this approach is advantageous optimization for very large systems and memory-constrained architectures, it typically requires more Hamiltonian applications to converge.

The residual minimization method with direct inversion in the iterative subspace improves convergence robustness by combining residual minimization with extrapolation techniques based on previous iterations. It is particularly useful for difficult electronic structure problems, such as systems with small band gaps or near degeneracies.
The parallel orbital update method emphasizes fine grained parallelism by allowing independent updates of individual orbitals. This reduces communication overhead and enables efficient scaling on very large parallel systems, although it may require more iterations to converge than subspace-based approaches.

Method Selection and Performance Considerations
The choice of an iterative eigensolver depends on the system characteristics and on the available computational resources. Davidson remains the most reliable general-purpose method, offering stable convergence and moderate memory requirements. The projected preconditioned conjugate gradient approach provides improved memory efficiency for large-scale problems, particularly on accelerator hardware. Residual minimization methods are preferred for difficult convergence scenarios, while parallel orbital update techniques are designed for extreme scale parallel environments.
Modern versions of Quantum ESPRESSO provide a unified interface for selecting these methods at runtime through input parameters. In practice, the Davidson method typically requires between five and fifteen iterations per self-consistent field cycle, with each iteration dominated by Hamiltonian applications. The GPU acceleration work presented in later chapters, therefore, focuses on optimizing the Davidson algorithm, since it remains the most widely used solver and its computational kernels are representative of the dominant operations in iterative diagonalization.

3 Diagonalization and Computational Methods

This chapter examines the computational aspects of diagonalization within Quantum ESPRESSO, encompassing the iterative solution of the Kohn-Sham equations, subspace construction procedures, convergence characteristics, and the performance properties of the Davidson eigensolver. Understanding these computational foundations is essential for the optimization strategies developed in this thesis. The GPU underutilization problem described in the Preface is not simply a matter of insufficient parallelism within individual operations, it is a structural consequence of how the Davidson algorithm is organized with respect to k-points and how its dominant operations map onto the GPU hardware. Addressing it effectively requires understanding precisely where computational time is spent, why certain operations are amenable to GPU acceleration while others are not, and what mathematical properties of the problem enable concurrent execution.

3.1 Bloch's Theorem and k-Point Sampling

The electronic structure of crystalline materials is determined by the periodicity arrangement of the atomic lattice. This translational symmetry enables quantum mechanical treatment by converting an effectively infinite problem into computations performed within a single unit cell. Bloch's theorem provides the theoretical foundation for this simplification, asserting that electronic states in periodic systems can be characterized by a crystal momentum quantum number. This section constructs the theoretical framework for k-point sampling within density functional theory calculations and confirms the computational decoupling of different k-points, a property serving as the foundation for parallelization approaches developed in later chapters.

3.1.1 Bloch's Theorem

This theorem states that for periodic potential, the eigenstates of the single-particle Hamiltonian can be chosen to have the form [Kit05]:

$$\psi_{nk}(r) = e^{ik\cdot r}u_{nk}(r) \tag{3.1}$$

where k is the crystal momentum, n is the band index, and $u_{nk}(r)$ is a function with periodicity of the lattice:

$$u_{nk}(r + R) = u_{nk}(r) \tag{3.2}$$

Bloch's theorem has very important computational implications. Rather than solving wavefunctions throughout an infinite crystal, the problem reduces finding the cell-periodic function $u_{nk}(r)$ within a single unit cell for each value of the quantum number k. For each k, there exist multiple solutions to the Schrödinger equation, indexed by the band number n.

The energies ϵ_{nk} form continuous functions of k called energy bands. The collection of

all band energies throughout the Brillouin zone constitutes the electronic band structure, which determines whether a material is metallic, semiconducting, or insulating.

3.1.2 k-Point Sampling in Density Functional Theory

In density functional theory for periodic systems, k-point sampling is the procedure of replacing the continuous integration over the Brillouin zone by a discrete sum over selected k-points to compute quantities such as total energy, charge density, and forces. The most widely used k-point generation scheme is the Monkhorst-Pack method in Quantum ESPRESSO, which constructs a uniform mesh:

$$k_{ijk} = \frac{2i - n_1 - 1}{2n_1} b_1 + \frac{2j - n_2 - 1}{2n_2} b_2 + \frac{2k - n_3 - 1}{2n_3} b_3 \tag{3.3}$$

for $i = 1, \ldots, n_1$, $j = 1, ., n_2$, $k = 1, \ldots, n_3$. An $n_1 n_2 n_3$ grid generates $N_k = n_1.n_2.n_3$ k-points distributed uniformly across the Brillouin zone. The required k-point density depends strongly on the material and property being calculated. Metals, with their sharp features at the Fermi surface, typically require dense meshes to achieve convergence of the total energy to within a few meV. Semiconductors and insulators, having band gaps and smoother density distributions, converge with coarser meshes.
Crystal symmetries significantly reduce the computational burden. Point group operations of the crystal map k-points onto equivalent positions in the Brillouin zone. Time-reversal symmetry relates to k and $-k$ for non-magnetic systems. These symmetries partition the Brillouin zone into irreducible wedges; only k-points within the irreducible Brillouin zone need explicit calculation.

3.1.3 Computational Independence of k-points

A crucial property for the parallelization strategies developed in this thesis is the computational independence of different k-points. For any two distinct k-points k_1 and k_2, the eigenvalue problems are completely decoupled:

$$\hat{H}(k_1)\psi_n(k_1) = \epsilon_n(k_1)\hat{S}(k_1)\psi_n(k_1) \tag{3.4}$$

$$\hat{H}(k_2)\psi_n(k_2) = \epsilon_n(k_2)\hat{S}(k_2)\psi_n(k_2) \tag{3.5}$$

These equations share no common terms and can be solved independently. Independence permits embarrassingly parallel execution using OpenMP threading. This k-point parallelization requires communication only at the end of each self-consistent-field iteration, when band energies and wavefunctions are gathered to construct the electron density.
The k-point independence property provides the theoretical foundation for the CPU parallelization strategy developed in Chapter 4 and for the GPU stream management approach in subsequent chapters. As different k-points can execute concurrently without synchronization, multiple CPU threads can invoke the Davidson solver simultaneously, and multiple CUDA streams can process k-points on the GPU in parallel. The following

section develops the mathematical structure of the eigenvalue problem itself and introduces the Davidson algorithm used to solve it.

3.2 The Kohn-Sham Eigenvalue Problem

The self-consistent solution of the Kohn-Sham equations requires repeated diagonalizations of the Hamiltonian operator to obtain electronic wavefunctions and energies. This section develops the mathematical structure of this eigenvalue problem, explains how the continuous differential equations are discretized into finite-dimensional matrix form using plane-wave basis sets, and establishes why iterative solution methods are essential for computational tractability.

3.2.1 From Kohn-Sham Equations to Matrix Form

The Kohn-Sham equations, discussed in section 2.3.2, are the building blocks of density functional theory. These equations must be solved for each k-point in the Brillouin zone to obtain the occupied states that determine the electron density. According to Bloch's theorem, any lattice-periodic function can be expressed as:

$$u_{nk}(r) = \sum_{G} c_{nk}(G) e^{iG\cdot r} \tag{3.6}$$

where the sum extends over all reciprocal lattice vectors G. The complete Bloch theorem is as follows:

$$\psi_{nk}(r) = \frac{1}{\Omega} \sum_{G} c_{nk}(G) e^{i(k+G)\cdot r} \tag{3.7}$$

where Ω is the unit cell volume and the normalization factor ensures proper quantum mechanical normalization. In practice, this infinite sum must be truncated. The kinetic energy of a plane wave with wavevector $k + G$ is:

$$\frac{\hbar^2}{2m_e} |k + G|^2 < E_{cut} \tag{3.8}$$

This defines a finite set of plane waves for each k-point, with the number of plane waves $n_{pw}(k)$ typically ranging from several thousand to tens of thousands depending on the cutoff energy and system size.
By substituting the plane-wave expansion into the Kohn-Sham equations and projecting onto the basis functions $e^{i(k+G).r}$ transforms the differential eigenvalue equation into an algebraic matrix eigenvalue problem:

$$\sum_{G} H_{GG'}(k) c_{nk}(G) = \epsilon_{nk} \sum_{G} S_{GG'}(k) c_{nk}(G) \tag{3.9}$$

In compact matrix notation:

$$H(k)c_{nk} = \epsilon_{nk}S(k)c_{nk} \tag{3.10}$$

This is a generalized eigenvalue problem of dimension $n_{pw}(k)$ x $n_{pw}(k)$.
The matrix formulation converts the problem from infinite-dimensional functional space to finite dimensional linear algebra. The dimension n_{pw}, while finite, remains large enough (typically $10^3 - 10^5$) that direct solution methods are computationally prohibitive. Moreover, the Hamiltonian matrices at different k-points are independent (Section 3.1.3), meaning that for calculations with N_k kpoints, N_k separate matrix eigenvalue problems must be solved at each self-consistent field iteration.

3.2.2 Structure of Hamiltonian Matrix

The Hamiltonian matrix elements are computed from:

$$H_{G'G}(k) = \frac{1}{\Omega}\int_{\Omega} e^{-i(k+G')\cdot r}\left[-\frac{\hbar^2}{2m_e}\nabla^2 + V_{eff}(r)\right]e^{i(k+G)\cdot r}d^3r \tag{3.11}$$

This integral separates contributions from different terms in the effective potential. The kinetic energy operator is diagonal in the plane-wave basis:

$$H^{kin}_{G'G}(k) = \frac{\hbar^2}{2m_e}|k+G|^2\delta_{G'G} \tag{3.12}$$

This diagonal structure depicts that plane waves are eigen functions of the kinetic energy operator. The local part of the effective potential which includes Hartree potential and local exchange correlation potential contributes off-diagonal terms:

$$H^{loc}_{G'G}(k) = V_{loc}(G'-G) \tag{3.13}$$

where $V_{loc}(G)$ is the Fourier transform of the real-space local potential:

$$V_{loc}(G) = \frac{1}{\Omega}\int_{\Omega} V_{loc}(r)e^{-iG\cdot r}d^3r \tag{3.14}$$

The local potential matrix is dense, meaning that most elements are nonzero, because the real-space potential $V_{loc}(r)$ varies on the atomic scale and thus couples many Fourier components. In practice, the local potential is not Fourier transformed directly. Instead, the application of the local potential to a wave function is performed in real space using Fast Fourier transforms (FFTs).
This FFT-based approach scales as $O(n_{pw} log n_{pw})$, rather than the $O(n^2_{pw})$ of explicit matrix multiplication, providing substantial computational savings. The non-local part of the pseudopotential, arising from angular momentum projections in the pseudopotential construction, takes the separable Kleinman-Bylander form:

$$H^{nl}_{G'G}(k) = \sum_{\alpha,l,m} D^{lm}_{\alpha}\beta_{\alpha,lm}(k+G')^*\beta_{\alpha,lm}(k+G) \tag{3.15}$$

Where α indexes atoms, l, m represents angular momentum quantum numbers, $\beta_{\alpha,lm}$ are projector functions, and $D_{\alpha^{lm}}$ are coupling coefficients determined by the pseudopotential. The projectors have the form:

$$\beta_{\alpha,lm}(k+G) = \int e^{-i(k+G)\cdot r}\beta_{\alpha,lm}(r)d^3r \tag{3.16}$$

The non-local contribution has low rank and is applied effectively through projector operations rather than explicit matrix formation. Application of the non-local potential to a wavefunction involves computing projections $<\beta_{\alpha,lm}|\psi>$, multiplying by coefficients D_α^{lm}, and adding back contributions $\sum_{\alpha,l,m} D_{\alpha lm}|\beta_{\alpha,lm}\rangle\langle\beta_{\alpha,lm}|\psi\rangle$.
This matrix-free approach is essential for handling large systems where explicit storage of $n_{pw}(k)$ x $n_{pw}(k)$ complex matrices would exceed available memory. The Hamiltonian matrices are Hermitian as required for quantum mechanical observables, ensuring real eigenvalues and orthogonal eigenvectors. This Hermitian structure is exploited by iterative eigen solvers, which can operate efficiently with only the matrix-vector product operation rather than requiring explicit matrix elements.

3.2.3 The Overlap Matrix

For norm-conserving pseudopotentials, the plane-wave functions $e^{i(k+G).r}$ forms an orthonormal basis, resulting in an identity overlap matrix:

$$S_{G'G}^{NC} = \delta_{G'G} \tag{3.17}$$

and reducing Eq. (2.10) to a standard eigenvalue problem. However, ultrasoft pseudopotential, which permits smoother pseudo-wavefunction, introduces a non-trivial overlap matrix. Ultrasoft pseudopotential augment the electron density with localized atomic contributions:

$$n(r) = \sum_{nk} f_{nk}|\psi_{nk}(r)|^2 + \sum_{\alpha,ij} Q_\alpha^{ij}(r)\rho_\alpha^{ij} \tag{3.18}$$

Where $Q_\alpha^{ij}(r)$ are augmentation functions centered on atom α, ρ_α^{ij} are occupation matrix elements. This augmentation modifies the normalization condition, leading to a generalized eigenvalue problem with overlap matrix:

$$S_{GG'}(k) = \delta_{GG'} + \sum_{\alpha,ij} \hat{Q}_{\alpha,ij}(G'-G)\langle\psi_{nk}|\beta_{\alpha,i}\rangle\langle\beta_{\alpha,j}|\psi_{nk}\rangle \tag{3.19}$$

Where $Q_\alpha^{ij}(G)$ are Fourier transforms of the augmentation functions. The overlap matrix is positive definite and Hermitian, ensuring well-posedness of the eigenvalue problem. However, its presence complicates the iterative solution compared to the norm-conserving case. Iterative methods must account for matrix overlap during the orthogonalization steps and in the convergence criteria, thereby increasing both computational cost and implementation complexity. For the ultrasoft pseudopotentials,

both H and S operations must be applied repeatedly during the Davidson iterations. This is reflected in the QE implementation through separate `h_psi` (Hamiltonian application) and `s_psi` (overlap application) subroutines, both of which are targets for GPU acceleration in Chapter 5.

3.3 Computational Implementation: Pseudopotentials, FFTs, and Dense Linear Algebra

The matrix-free implementation of the Hamiltonian described in Section 3.2.2 relies on three key computational techniques: pseudopotentials to reduce the number of electrons, fast Fourier transforms to efficiently apply the local potential, and dense linear algebra for subspace operations.

3.3.1 Pseudopotentials: Reducing Computational Complexity

The all-electron problem:
In principle, the Kohn-Sham equations should be solved for all electrons in the system—both valence and core electrons. For example, for a silicon atom, this means treating all 14 electrons: $1s^2\ 2s^2\ 2p^6\ 3s^2\ 3p^2$. The core electrons $(1s, 2s, 2p)$ occupy tightly bound orbitals close to the nucleus, exhibiting rapid spatial oscillations. Representing these oscillations in a plane-wave basis requires extremely high energy cutoffs—effectively, very short-wavelength plane waves.
The Pseudopotential approximation: Core electrons participate minimally in the chemical bonding—their wavefunctions and charge distributions remain nearly identical whether the atom is isolated or part of a crystal. This observation motivates the pseudopotential approximation: replacing the strong Coulomb potential of the nucleus and the core electrons with an effective, weaker potential that acts only on valence electrons.
The pseudopotential $V_{ps}(r)$ is constructed to reproduce the valence properties of the all-electron potential but without the rapid oscillations near the nucleus [Pic89]. The pseudo-wavefunctions of valence electrons are smooth, requiring far fewer plane waves to be represented accurately.
Types of pseudopotentials:

1. **Norm-conserving pseudopotentials (NCPP)**: Constructed such that the charge within a chosen core radius equals that of the all-electron wavefunction [HSC79]. These pseudopotentials yield an overlap matrix $S = I$ (identity), simplifying the eigenvalue problem. However, they require moderate energy cutoffs (40-80 Ry for first-row elements of the periodic table, meaning elements like Carbon, Nitrogen, Oxygen, Fluorine).

2. **Ultrasoft pseudopotentials (USPP)**: Introduced by Vanderbilt (1990) [Van90], these relax the norm-conservation constraint to achieve even smoother

pseudo-wavefunctions. The trade-off is a non-trivial overlap matrix $S \neq I$, requiring solution of the generalized eigenvalue problem $H\psi = \epsilon S\psi$ (Section 2.2.3). USPPs enable lower energy cutoffs (25-40 Ry) at the cost of increased algebraic complexity.

3.3.2 Fast Fourier Transforms: Applying the Local Potential

The local part of the effective potential comprises the Hartree potential and the exchange-correlation potential, which must be applied to the wavefunctions during each Hamiltonian evaluation. This operation is a matrix-vector product:

$$\langle \vec{G} | \hat{V}_{loc} | \psi \rangle = \sum_{\vec{G}'} V_{loc}(\vec{G} - \vec{G}') \psi_{\vec{G}'} \tag{3.20}$$

Where G and $\vec{G}$ are reciprocal lattice vectors. For n_{pw} plane waves, this is an n_{pw} x n_{pw} dense matrix whose storage is prohibitive in memory and impossible to compute efficiently.
For example : small silicon test case with $E_{cut} = 14.0$ Ry, $n_{pw} \approx 500$, the matrix would require $500^2 \times 16 = 4$ MB per k-point, which while storable becomes computationally prohibitive when repeated across thousands of SCF iterations. Instead, the operation is performed efficiently using the Fast Fourier Transform (FFT), which evaluates the equivalent convolution in $O(n_{pw} \log n_{pw})$ operations rather than $O(n_{pw}^2)$.
Real-space multiplication via FFT:
The key insight is that $V_{loc}(r)$ is diagonal in real space: multiplying the potential by a wavefunction at each spatial point is trivial. The convolution theorem of Fourier analysis states that multiplication in real space corresponds to convolution in reciprocal space [Pay+92]. Therefore, applying the local potential can be decomposed into three steps:

1. **Inverse FFT**: Transform wavefunction from reciprocal space to real space:

$$\psi_{nk}(\vec{r}) = \sum_{\vec{G}} c_{nk}(\vec{G})\, e^{i\vec{G}\cdot\vec{r}} \tag{3.21}$$

2. **Pointwise multiplication**:Multiply by potential in real space:

$$\phi_{nk}(\vec{r}) = V_{loc}(\vec{r}) \cdot \psi_{nk}(\vec{r}) \tag{3.22}$$

3. **Forward FFT**: Transform result back to reciprocal space:

$$(V_{loc}\psi)_{nk}(\vec{G}) = \sum_{\vec{r}} \phi_{nk}(\vec{r})\, e^{-i\vec{G}\cdot\vec{r}} \tag{3.23}$$

The FFT-based approach scales as $O(n_{grid} \log n_{grid})$ compared to $O(n_{pw}^2)$ for explicit matrix multiplication, representing a dramatic reduction in computational cost. For a typical medium-sized system with $E_{cut} = 60$ Ry and $n_{pw} \sim 10^4$ plane waves, the real-space grid has $n_{grid} \sim 8n_{pw}$, giving approximately 8×10^4 grid points. The FFT-based approach

then requires:

$$n_{grid} \log_2 n_{grid} \approx 8 \times 10^4 \times 17 \approx 1.4 \times 10^6 \text{ operations} \tag{3.24}$$

compared to the direct matrix multiplication cost of:

$$n_{pw}^2 = (10^4)^2 = 10^8 \text{ operations} \tag{3.25}$$

a reduction of nearly two orders of magnitude. For our silicon test case with $E_{cut} = 14.0$ Ry and $n_{pw} \approx 500$, the FFT requires only $\sim 4{,}500$ operations per application compared to 250,000 for direct multiplication, already a 55× improvement even at this small scale. .
FFT implementation:
Modern FFT libraries (FFTW, Intel MKL, cuFFT on GPUs) provide highly optimized implementations exploiting cache hierarchies, SIMD vectorization, and parallel execution. The FFT operations constitute the primary source of memory bandwidth demand in the Davidson algorithm—each FFT requires reading and writing all n_{pw} complex numbers (16 bytes each), stressing memory subsystems. The combination of high computational complexity, memory-bandwidth sensitivity, and regular computational structure makes FFTs excellent candidates for GPU acceleration.

3.3.3 BLAS for Linear Algebra

Density functional theory calculations based on plane-wave basis sets require extensive dense linear algebra operations to construct and manipulate electronic wavefunctions, compute overlap integrals, and perform basis transformations. These operations are implemented using the Basic Linear Algebra Subprograms (BLAS) library, a standardized collection of routines providing highly optimized implementations of fundamental vector and matrix operations.
BLAS Operation Hierarchy
BLAS routines are organized into three levels based on computational complexity and data structures:
Level 1 BLAS performs vector operations such as dot products ($x^T y$) and vector scaling ($y \leftarrow \alpha x$). These operations require $O(n)$ floating point operations and $O(n)$ memory access, which resulted in an arithmetic intensity of ≈ 1:1. On modern processors where memory bandwidth limits performance, Level 1 BLAS is memory bound.
Level 2 BLAS performs matrix-vector operations such as $y \leftarrow \alpha Ax + \beta y$. For an m×n matrix, these operations require $O(mn)$ work with similar memory access counts, maintaining low arithmetic intensity (≈2:1), which is largely memory-bound.
Level 3 BLAS performs matrix-matrix operations, most importantly the general matrix multiplication "ZGEMM" [Don+90]:

$$C \leftarrow \alpha AB + \beta C \tag{3.26}$$

For nn matrices, these operations require $2n^3$ floating point operations while accessing only $3n^2$ matrix elements, which resulted into arithmetic intensity of $O(n)$. For large matrices, this high ratio makes Level 3 BLAS compute-bound, enabling near-peak

computational performance.

BLAS in Iterative Diagonalization

The Davidson algorithm relies heavily on BLAS operations, with ZGEMM dominating the computational workload. The primary uses include:

Subspace projection: Constructing the reduced Hamiltonian requires computing overlaps between basis functions and their Hamiltonian-transformed counterparts. For a subspace of dimension m with n_{pw} plane-wave coefficients per function, this projection is efficiently performed via ZGEMM with complexity $O(n_{pw}m^2)$.

Basis transformation: Converting between the full plane-wave representation and reduced subspace representation involves matrix-matrix multiplications of the form $|\phi_{new}\rangle = \sum_i c_i|\psi_i\rangle$ where coefficients c_i come from subspace diagonalization. This transformation via ZGEMM requires $O(n_{pw}m^2)$ operations.

Wavefunction orthonormalization : Ensuring orthogonality $\langle\psi_i|\psi_j\rangle = \delta_{ij}$ across multiple functions is implemented via ZGEMM rather than individual dot products, improving cache efficiency and reducing overhead. These operations account for 30-40% of total Davidson iteration time, making BLAS a primary optimization target alongside FFT operations. Modern BLAS implementations achieve high performance through several optimizations, including cache blocking, vectorization, and threading, by exploiting SIMD vector instructions and multi-threading.

Library Implementation: Quantum ESPRESSO links against optimized BLAS libraries such as Intel MKL or OpenBLAS, which achieve 70-90% of theoretical peak CPU performance for large matrix operations.

GPU acceleration potential: Level 3 BLAS operations are ideally suited for GPU offloading due to high arithmetic intensity ($O(n^3)$operations on $O(n^2)$ data), regular memory access patterns enabling coalesced memory transactions, and massive parallelism—each resulting element computable independently across thousands of GPU threads. The NVIDIA cuBLAS library achieves 85-95% of theoretical peak GPU performance for ZGEMM, providing substantial computational throughput advantages over CPU implementations.

In summary, BLAS operations constitute a critical component of iterative diagonalization, consuming significant computational resources while exhibiting characteristics that make them excellent targets for both multi-threading and GPU acceleration

3.4 Computational Cost Analysis

The Davidson diagonalization algorithm, introduced in Section 2.4.2, exhibits distinct computational patterns that determine its suitability for parallelization and hardware acceleration. Understanding where computational time is spent, how memory is utilized, and what parallelization opportunities exist is essential for designing effective optimization strategies. This section analyses these characteristics to establish the foundation for the CPU and GPU acceleration techniques developed in Chapters 4 and 5.

3.4.1 Cost Distribution in Davidson Iterations

Each Davidson iteration consists of several computational operations with distinct characteristics. The most expensive operations are the matrix-vector products—applying the Hamiltonian and overlap operators to basis vectors ($H|v\rangle$ and $S|v\rangle$). These operations involve fast Fourier transforms for the local potential, projector applications for the non-local pseudopotential, and kinetic energy terms. The second major contribution comes from the subspace projection operations, where the inner products$\langle v_i|H|v_j\rangle$ are computed to construct the reduced Hamiltonian and the overlap matrices. These operations are implemented using dense linear algebra routines (BLAS), particularly matrix-matrix multiplications (ZGEMM).
The dominance of matrix-vector products and dense linear algebra operations indicates that these components are the primary targets for optimization through parallelization and GPU acceleration.

3.4.2 Parallelization Opportunities

The Davidson algorithm presents parallelization opportunities at multiple levels, with k-point parallelism being the most effective approach for electronic structure calculations.

k-Point parallelism: As established in Section 3.1.3, eigenvalue problems at different k-points are mathematically independent. This independence enables straightforward parallel execution, in which each k-point is assigned to a separate computational thread or GPU stream. The k-points can be processed in parallel, with no communication required during the diagonalization phase, thereby representing an embarrassingly parallel workload. Communication is required only at the end of each SCF iteration, when results from all k-points are gathered to construct the electron density, with negligible overhead relative to diagonalization time. This parallelization strategy achieves near-perfect scaling efficiency when the number of k-points equals or exceeds the number of parallel execution units (CPU threads or GPU streams). For calculations with multiple k-points, this approach provides the most straightforward path to performance improvement and forms the basis for the CPU parallelization strategy developed in Chapter 4.

Within-k-point parallelism: For individual k-point diagonalizations, parallelism exists within the dense linear algebra operations. Modern BLAS libraries automatically exploit multi-threading for matrix-matrix multiplication, yielding moderate speedups for subspace projection operations. However, the overall benefit is limited because these operations represent only a fraction of the total iteration cost.

GPU acceleration: Graphics processing units offer massive thread-level parallelism well-suited to the operations in the Davidson diagonalization. FFT operations, dense matrix multiplications, and element-wise vector operations all map efficiently to GPU architectures. The primary challenge lies not in parallelizing individual operations but in achieving concurrent execution of multiple k-point diagonalizations on the GPU to fully

utilise the computational resources.
The combination of k-point parallelism and operation-level acceleration (exploiting parallelism within FFT and BLAS operations) provides a path to significant performance improvements on modern multi-core CPUs and GPU-accelerated systems. The subsequent chapters detail the implementation and performance of these optimization strategies.

3.5 Summary and Motivation

This chapter presented the theoretical foundations underlying the optimization strategies in this thesis. Bloch's theorem demonstrates that periodic crystal symmetry reduces electronic-structure calculations to solving eigenvalue problems at discrete k-points in the Brillouin zone. The crucial property established is that k-points are mathematically independent—they can be solved simultaneously without communication, enabling embarrassingly parallel execution. The Kohn-Sham eigenvalue problem was formulated in the plane-wave representation, which explains why iterative methods like Davidson are essential: direct diagonalization scales as $O(n^3)$, whereas iterative methods achieve $O(n^2)$ by computing only the lowest eigenvalues needed for the electron density construction. A Computational cost analysis identified matrix-vector products (dominated by FFTs and projector operations) and dense linear algebra (BLAS operations) as the primary time consumers in Davidson iterations, both of which are well-suited for multi-threading and GPU acceleration. The k-point independence property, combined with the parallelizable nature of the dominant operations, establishes Davidson diagonalization as an ideal optimization target. These characteristics motivate the two complementary strategies developed in the subsequent chapters for addressing GPU underutilization in small-sized Quantum ESPRESSO jobs: CPU-level k-point batching via OpenMP threading (Chapter 4), which exploits the mathematical independence of k-points to process multiple Davidson solution concurrently on the host, and GPU-level concurrent stream execution via OpenACC (Chapter 5), which assigns each concurrent k-point its own CUDA stream so that their GPU operations overlap and fill the idle gaps that sequential execution would leave.

4 CPU Parallelization via k-Point Batching

The theoretical framework presented in Chapter 3 confirmed the mathematical independence of k-points in electronic-structure calculations, in which each k-point is treated as a separate Hamiltonian, with no inter-k-point coupling in the eigenvalue problem. This intrinsic independence offers an excellent opportunity for parallelization, enabling concurrent diagonalization of multiple k-points without inter-thread communication. This chapter outlines the k-point batching strategy implemented with OpenMP threading to exploit parallelism, examines the challenges of maintaining thread safety, and evaluates the achieved performance gains.

4.1 Motivation and Strategy

The computational analysis in Section 3.4 identified the Davidson diagonalization algorithm as the primary performance bottleneck in self-consistent field calculations, typically consuming between 60% and 80% of the total execution time. Within a single SCF iteration, the Davidson solver must operate on each k-point throughout the irreducible Brillouin zone. The serial execution pattern processes these k-points sequentially:

```
1  DO ik = 1, nks
2     CALL davidson_solver(ik, eigenvalues, eigenvectors)
3  END DO
```

Such sequential execution leads to substantial underutilization of contemporary multi-core processors. On a workstation with 8 to 16 cores, the Davidson solver runs on a single core, while the remaining cores remain unused. Given the mathematical independence of k-points (Section 3.1.3), a straightforward parallelization strategy emerges: distributing k-points across available CPU cores through OpenMP threading, allowing each thread to solve its assigned eigenvalue problem independently.

The expected performance gain is approximately linear with respect to the number of threads, limited only by the number of k-points. For computations with N_k k-points running on N_{threads} cores, the theoretical speedup equals $\min(N_{\text{threads}}, N_k)$. This embarrassingly parallel configuration should achieve parallel efficiencies exceeding 90%, significantly superior to alternative parallelization methods requiring inter-thread communication.

Implementing this parallelization directly in the full Quantum ESPRESSO codebase, which consists of hundreds of interconnected source files forming a complex software ecosystem, would have been impractical. The extensive nature of the complete PWscf code significantly hinders effective debugging and validation of parallel enhancements. As a result, we pursued a miniapp development strategy: separating the Davidson solver and its essential dependencies into a self-contained, streamlined driver program. This

extraction maintains the critical computational properties while keeping the codebase manageable and accessible.

In addition to improving utilization of multi-core CPU resources, k-point batching addresses a more fundamental challenge that arises in GPU-accelerated calculations for small systems. When k-points are processed sequentially on a Graphics Processing Unit (GPU), each individual Davidson diagonalization provides insufficient computational work to fully utilize the device, since the small plane-wave basis and few bands characteristic of small-size jobs result in matrix dimensions and operation counts that are too small to engage all streaming multiprocessors of a modern GPU simultaneously. The batching framework developed in this chapter, therefore, serves not only as a strategy for CPU-level parallelism but also as a necessary structural foundation for enabling GPU concurrency, as discussed in Chapter 5. Without the simultaneous execution of multiple k-points, there is no possibility for overlapping their associated GPU operations.

4.2 The cb_davidson Miniapp

4.2.1 Rationale for Miniapp Development

The full QE package provides extensive functionality beyond the Davidson eigensolver, including structure optimization, molecular dynamics, phonon calculations, response properties, and numerous post-processing tools. The Davidson solver (cegterg.f90) is embedded in this ecosystem and called from the main SCF driver, which in turn is part of the larger PWscf workflow. Attempting to parallelize k-point execution within this full context introduces multiple challenges:

1. **Code complexity:** Understanding all dependencies and data flow paths requires substantial effort. Unintended interactions with other code sections could mask threading issues.

2. **Build complexity:** QE's build system involves multiple libraries, conditional compilation, and platform-specific configurations. Isolating compilation issues from threading issues becomes difficult.

3. **Debugging difficulty:** Parallel bugs (race conditions, deadlocks) are notoriously hard to diagnose. A smaller codebase reduces the search space for problems.

4. **Development speed:** Testing modifications in the full QE requires compiling thousands of source lines. A miniapp compiles in seconds, enabling rapid iteration.

The miniapp development approach, well-established in the HPC community, provides an effective response to these obstacles [Gia+09]. Isolating the Davidson solver, along with essential support functions, into a compact, standalone program enabled the creation of a controlled development environment for the parallel k-point execution strategy. Upon successful validation in this reduced-complexity setting, the implementation could be reliably transitioned to the complete software system.

4.2.2 Miniapp Structure

The cb_davidson mini application is structured as a set of subdirectories within the KS_SOLVER_CB repository, with each component providing a specific layer of functionality [Nis24]. Rather than replicating existing Quantum ESPRESSO library code, the mini application directly relies on the same supporting libraries that underpin the full PWscf implementation, including LAXlib, FFTXlib, DEVMEMLIB, and UtilXlib. This design ensures that performance measurements obtained from the mini application accurately reflect realistic production-level execution conditions. At the highest level, the repository contains a main directory that organizes the solver code, supporting libraries, example inputs, and build configuration files. The cbToy subdirectory contains the principal mini-application source files. The driver program `cb_davidson_main.f90` controls the outer k-point iteration, manages OpenMP parallel regions, and initializes GPU execution streams. The Davidson solver implementation, `cegterg.f90(/task_parallelization_gpu/KS_Solvers/Davidson)`, is derived from the original QE solver and has been extended to support batch indexing and GPU stream management.
Additional routines within this directory implement the main operator applications required by the Davidson algorithm. Specifically, the files `cb_h_psi.f90`, `cb_s_psi.f90`, and `cb_g_psi.f90` perform the Hamiltonian application, overlap matrix application, and preconditioning operations, respectively. Each of these routines accepts a batch index parameter to select the appropriate portion of the batched workspace arrays. The test input `examples/si2_11_points.in` describes a silicon system with 11 k-points and representative parameters, including ecutwfc = 14.0 Ry and nk_batches, which is specifiable at run time. The total wall time for the serial CPU baseline is approximately 2.8 seconds, providing a computationally non-trivial yet rapidly executable test case suitable for iterative development and benchmarking.

4.3 OpenMP Implementation of k-Point Batching

4.3.1 Parallel Region Design

The k-point parallelization employs a two-level loop structure: an outer serial loop iterating over k-point batches, and an inner parallel loop where OpenMP threads process individual k-points concurrently. This batching strategy manages memory consumption by processing k-points in groups rather than loading all k-points simultaneously. Rather than parallelizing across all k-points at once, k-points are processed in batches of size nk_batches. For a calculation with 11 k-points and batch size 4:

- Batch 1: k-points 1-4 processed in parallel (4 threads)
- Batch 2: k-points 5-8 processed in parallel (4 threads)
- Batch 3: k-points 9-11 processed in parallel (3 threads)

The workspace arrays are dimensioned (`npwx`, `nbnd`, `nk_batches`), where `npwx` is the maximum number of plane waves across all k-points (an upper bound on n_{pw}, the number

of plane waves at a given k-point, which varies slightly between k-points due to the spherical cutoff in reciprocal space), and nbnd is the number of electronic bands included in the calculation, memory usage remains bounded at approximately 2×nk_batches GB regardless of total k-point count. It only holds because k-points are processed in batches not all stored simultaneously.

Implementation Structure: The parallelization in cb_davidson_main.f90 follow this pattern:

```
! Pre-allocate batched workspace (3D arrays)
allocate(evc_batched(npwx, nbnd, nk_batches))
allocate(eig_batched(nbnd, nk_batches))

! Outer serial loop over batches
do ik = 1, nks, nk_batches

   !$omp parallel num_threads(nk_batches)
   do i_batch = 1, min(nk_batches, nks - ik + 1)

      clock_thread = i_batch
      current_k = ik + i_batch - 1

      call init_k(current_k, i_batch)
      call cegterg(..., evc_batched(:,:,i_batch), ..., i_batch)

   end do
   !$omp end parallel

   ! Sequential output processing
   do i_batch = 1, min(nk_batches, nks - ik + 1)
      call write_bands(eig_batched(:,i_batch), ...)
   end do
end do
```

Key design features: Two-level loop structure: The outer loop advances by nk_batches at each iteration (ik = 1, 5, 9, ... for batch size 4), ensuring only one batch executes at a time. The inner loop bound min (nk_batches, nks - ik + 1) handles incomplete final batches:when ik = 9 with nks = 11, only 3 iterations execute rather than 4.

Thread-to-k-point mapping: Each thread calculates its global k-point index via current_k = ik + i_batch - 1. The thread executing i_batch = 3 in the first outer iteration (ik = 1) processes global k-point 3. This mapping distributes k-points across threads without an explicit assignment logic.

Pre-allocated batched arrays: Workspace is allocated once before the k-point loop with a third dimension indexed by batch position. Each thread accesses its designated slice: thread processing i_batch = 2 uses evc_batched(:,:,2), eig_batched(:,2), etc. This design eliminates dynamic allocation overhead within the parallel regions while preventing memory conflicts—each thread's data occupies non-overlapping memory.

Batch-specific routine arguments: The Davidson solver and supporting routines accept an additional i_batch parameter that identifies which batch slice to use:

```
call cegterg(..., evc_batched(:,:,i_batch), ..., i_batch)
call my_h_psi_batched(..., fft_array_batched(:,i_batch), ...)
```

This index enables thread-safe access to shared workspace arrays—all threads reference the same evc_batched array but access different slices along the third dimension.

Two-pass processing: Computation and output are separated into distinct phases. The first parallel loop performs Davidson diagonalization concurrently, with threads working independently. After an implicit barrier at the end of the parallel region, a second sequential loop writes the results. This separation ensures deterministic output ordering (k-points 1, 2, 3, ...) and avoids I/O contention from simultaneous file access.

This implementation successfully distributes the embarrassingly parallel k-point workload across OpenMP threads while maintaining bounded memory consumption and deterministic ordering.

4.3.2 Data Management and Scope

The correct specification of a variable scope is critical for thread safety. Variables are classified as either thread-private, which means each thread has its own copy, or shared, that is a single copy is accessed by all threads.

Thread-private variables:

```
!$omp parallel ... private(i_batch)
```

The loop index i_batch is explicitly private means each thread maintains its own value. Additional thread-private variables include current_k and clock_thread, plus any variables declared within the parallel region.

Shared variables with thread-specific indexing:

```
!$omp parallel ... default(shared) ... shared(...)
```

The default(shared) clause makes variables shared unless explicitly privatised. The critical design principle is that while arrays like evc_batched are shared, each thread accesses non-overlapping elements via batch indexing:

- evc_batched(:,:,1) accessed only by thread with i_batch = 1
- evc_batched(:,:,2) accessed only by thread with i_batch = 2

This indexed access pattern is safe without locking because no memory location is accessed simultaneously by multiple threads. The third array dimension effectively partitions the workspace among threads.

4.3.3 Runtime Thread Control

The number of OpenMP threads is configured at runtime via the input-file parameter nk_batches, following the standard Quantum ESPRESSO namelist convention. This parameter is specified in the &system namelist:

```
&system
   crystal_name='Si', ecutwfc=14.d0, ethr=1.d-6,
   ncell=2, david=9, nk_batches=2
/
```

The nk_batches value is read during input parsing and passed directly to omp_set_num_threads, controlling the size of the OpenMP thread pool for all subsequent parallel regions. This design follows the established Quantum ESPRESSO convention, in which all calculation parameters are specified in input files rather than command-line arguments.
Execution:
Same executable for all configurations

```
mpirun -np 1 ./cb_davidson.x < si2_11_points.in
```

To test different thread counts, only the input file requires modification — no recompilation is needed. The following example uses the SLURM workload manager, which is the job scheduler used on the Leonardo supercomputer.

```
# In job script - must match nk_batches in input file
export OMP_NUM_THREADS=2        # Match nk_batches=2
#SBATCH --cpus-per-task=2       # Match nk_batches=2
```

This approach provides several advantages. All the calculation parameters, including parallelization settings, are documented in a single input file, simplifying reproducibility and parameter tracking. Different configurations require only modifications to the input file, not changes to commands, consistently with the standard Quantum ESPRESSO workflows. Additionally, the input-file approach naturally integrates with job submission systems, where multiple input files with different nk_batches values can be submitted simultaneously.

4.4 Thread Safety Challenge: The Timing System

4.4.1 Initial Problem Manifestation

After implementing the parallel k-point loop described in Section 3.3, initial testing with multiple threads produced unexpected behaviour. While the eigenvalues appeared correct, the timing output contained errors:
Warning: clock 'davidson' already started Warning: clock 'cegterg' not running, cannot stop

These warnings suggested that the timing system, which is designed to measure the execution time of different sections of the code, was being used incorrectly in the parallel context.
The problem exhibited the classic race condition characteristics: it was intermittent and non-deterministic. Some runs completed without warnings; others produced errors. The precise outcome depended on the relative timing of thread execution, that is, which thread reached certain code points first, making the bug difficult to reproduce consistently.
Serial execution always produced correct, consistent timings. Only multi-threaded execution exhibited problems. This pattern strongly indicated that the timing system, originally written for serial code, contained a shared state that became corrupted under concurrent access from multiple threads.

4.4.2 Root Cause: Non-Thread-Safe Clocks

The timing infrastructure in clocks_handler.f90 was designed for serial execution. It provided the functions start_clock(label) and stop_clock(label) to bracket code sections and measure the elapsed time. The implementation stored timing data in module-level arrays:

```
1  ! Original (non-thread-safe) design
2  REAL(DP) :: t0cpu(maxclock)            ! Start times - one per clock
3  REAL(DP) :: t0wall(maxclock)           ! Wall times - one per clock
4  LOGICAL :: clock_running(maxclock)     ! Status flags
```

These arrays are global—only one copy exists in memory, shared by all the threads. When multiple threads simultaneously call ‘start_clock('davidson')‘, they all attempt to read and write the same memory locations, creating a race condition.

4.4.3 Clock Context Mismatch

In the original serial code, timing calls were placed at different nesting levels:

```
1  ! Problematic placement
2  CALL start_clock('davidson')      ! Outside parallel region (serial)
3
4  !$omp parallel
5     DO i_batch = 1, nk_batches
6        CALL cegterg(...)
7        CALL stop_clock('davidson')  ! Inside parallel region (multiple threads)
8     END DO
9  !$omp end parallel
```

The problem is that the clock was started once serially by the master thread but stopped multiple times by each thread independently, resulting in undefined measurements. The

possible solution is to keep the start and the stop in the same context, either both inside or both outside the parallel region.

4.4.4 Solution: Per-Thread Timing Architecture

The corrected timing system allocates separate storage for each thread, eliminating the race condition by spatially separating the data. Design principles are as follows:

1. **Two-dimensional timing arrays:** Instead of t0cpu(maxclock) storing one start time per clock, the arrays become t0cpu(max_threads, maxclock), storing one start time per thread per clock.

```
1      ! Thread-safe design
2      REAL(DP), ALLOCATABLE :: t0cpu(:,:)     ! (nthreads, maxclock)
3      REAL(DP), ALLOCATABLE :: t0wall(:,:)    ! (nthreads, maxclock)
```

2. **Thread identification via clock_thread:** A thread-private variable identifies which thread is executing:

```
1      INTEGER :: clock_thread = 0
2      !$omp threadprivate(clock_thread)
```

   Each OpenMP thread maintains its own independent copy of clock_thread.

3. **Context-aware initialization :** The timing system detects whether code is executed in a parallel or serial context:

```
1      ! In start_clock and stop_clock
2      if (omp_in_parallel()) then
3         mythread = omp_get_thread_num()
4         call omp_set_lock(clock_locker)
5      else
6        clock_thread = 1
7      end if
```

   For serial execution, clock_thread is always set to 1 to ensure backward compatibility. For parallel execution, the value set by the application is used.

4. **OpenMP locks for clock management:** While timing storage is done per thread independently, OpenMP locks protect the clock lookup and creation logic:

```
1      #if defined(_OPENMP)
2         INTEGER(kind=omp_lock_kind) :: clock_locker
3      #endif
```

   When multiple threads simultaneously call start_clock, the lock (omp_set_lock(clock_locker)) ensures only one thread at a time searches the clock_label array or creates new clocks. This prevents race conditions in clock

bookkeeping while keeping actual timing measurements independent. The lock is released (omp_unset_lock(clock_locker)) after the critical management operations complete.

Implementation in start_clock:

```
      ! Acquire lock for parallel clock management
      if (omp_in_parallel()) then
         mythread = omp_get_thread_num()
         call omp_set_lock(clock_locker)
      else
         clock_thread = 1
      end if

      ! Find or create clock (protected by lock)
      DO n = 1, nclock
         IF (clock_label(n) == label_) THEN
            ! Store start time using thread-specific index
            t0cpu(clock_thread, n) = f_tcpu()
            t0wall(clock_thread, n) = f_wall()

            if (omp_in_parallel()) call omp_unset_lock(clock_locker)
            RETURN
         ENDIF
      END DO

      ! Release lock after clock operations
      if (omp_in_parallel()) call omp_unset_lock(clock_locker)
```

Each thread stores its start time in t0cpu(clock_thread, n), where `clock_thread` identifies its unique row. Thread with clock_thread = 1 uses the first row, `clock_thread` `=` 2 uses the second row, ensuring no overlap.

Implementation in stop_clock:

```
! Acquire lock for clock management
if (omp_in_parallel()) then
   call omp_set_lock(clock_locker)
else
   clock_thread = 1
```

```
6  end if
7
8  ! Find clock and compute elapsed time
9  IF (clock_label(n) == label_) THEN
10    ! Compute elapsed time using thread's own start time
11    cputime(n) = cputime(n) + f_tcpu() - t0cpu(clock_thread, n)
12    walltime(n) = walltime(n) + f_wall() - t0wall(clock_thread, n)
13    called(n) = called(n) + 1
14
15    ! Mark this thread's clock as stopped
16    t0cpu(clock_thread, n) = notrunning
17    t0wall(clock_thread, n) = notrunning
18
19    if (omp_in_parallel()) call omp_unset_lock(clock_locker)
20    RETURN
21 ENDIF
```

Each thread reads from t0cpu(clock_thread, n) using the same clock_thread value it used when calling start_clock.

Accumulation of timing data: The shared variables cputime(n) and walltime(n) accumulate elapsed time across all threads:

```
1  cputime(n) = cputime(n) + f_tcpu() - t0cpu(clock_thread, n)
```

For example when Thread 1 adds 0.7 seconds and Thread 2 adds 0.8 seconds, cputime(n) becomes 1.5 seconds, representing aggregate computational work. The OpenMP lock ensures these additions don't conflict when multiple threads update simultaneously.

4.4.5 Consistent Clock Placement

Alongside the thread-safe storage, the clock placement was corrected to ensure the start and the stop calls occur in matching contexts:

```
1  ! Corrected: both calls inside parallel region
2  !$omp parallel num_threads(nk_batches)
3    CALL start_clock('davidson')
4    DO i_batch = 1, nk_batches
5      CALL cegterg(...)
6    END DO
7    CALL stop_clock('davidson')
8  !$omp end parallel
```

Each thread independently starts its clock, executes work, and stops its clock within the same parallel context. For serial sections:

```
! Both calls outside parallel region
CALL start_clock('total')
!$omp parallel
   ! Parallel work
!$omp end parallel
CALL stop_clock('total')
```

A single thread manages the clock before and after the parallel region, maintaining consistent context.

4.5 Performance Results

With the k-point batching implementation validated for correctness and the timing infrastructure verified as thread-safe, performance measurements quantified the speedup achieved through OpenMP parallelization. All measurements were performed on the 11 k-point silicon test case (si2_11) on the NVIDIA DGX Spark workstation, which is powered by the GB10 Grace Blackwell Superchip featuring a 20-core ARM CPU and 128 GB of unified LPDDR5X memory providing 273 GB/s memory bandwidth [NVI24d].

4.5.1 Execution Time and Speedup

All performance results reported in this section were obtained using the following software stack on the NVIDIA DGX Spark workstation. The code was compiled using the NVIDIA HPC SDK (NVHPC) 24.5 toolchain. Table 4.1 presents the wall-clock execution times for the complete Davidson diagonalization across 11 k-points, measured for thread counts of 1, 2, and 4.

Threads (nk_batches)	Wall Time (s)	Speedup	Parallel Efficiency
1 (serial baseline)	2.848	1.00x	100%
2	1.581	1.80x	90%
4	0.852	3.34	84%

Table 4.1: Execution time, speedup, and parallel efficiency for the si2_11 test case on DGX Spark.

The serial implementation provides a baseline execution time of 2.848 seconds. When 2 threads are applied, the runtime decreases to 1.581 seconds, representing a 1.80× acceleration and achieving 90% parallel efficiency. With 4 threads, the execution time is reduced to 0.852 seconds, delivering a 3.34× speedup alongside 84% parallel efficiency. The substantial parallel efficiency maintained across both thread counts validates that the k-point batching approach successfully leverages thread-level parallelism with negligible overhead.

4.5.2 Scaling Analysis

The measured speedup approaches ideal linear scaling but exhibits a slight deficit, indicating sublinear behaviour that is expected and tolerable for computations of this size. Parallel efficiency decreases from 90% with 2 threads to 84% with 4 threads, indicating a gradual deterioration that suggests that implementation continues to perform well with increasing thread counts. Several factors contribute to the deviation from ideal scaling:
Load imbalance: The 11 k-point test case does not divide evenly across 4 threads. The batching structure processes k-points as follows: batch 1 handles k-points 1-4 (4 threads active), batch 2 handles k-points 5-8 (4 threads active), and batch 3 handles k-points 9-11 (only 3 threads active, one idle). This uneven distribution causes one thread to sit idle during the final batch, reducing parallel efficiency by approximately 9%.
Memory bandwidth contention: The DGX Spark's unified memory architecture—where CPU and GPU share the same 128 GB LPDDR5X pool at 273 GB/s—means all CPU threads compete for the same memory bandwidth. The sublinear scaling likely results from memory bandwidth contention during parallel FFT execution. Since FFTs have low arithmetic intensity, they are memory bandwidth bound rather than compute bound. Consequently, as threads compete for data access, the effective bandwidth per thread drops and causes performance to stall. This contention is more pronounced in unified memory systems than in traditional CPU architectures with dedicated CPU memory subsystems.
Synchronization overhead: Each OpenMP parallel region introduces minor overhead from thread coordination and barrier synchronization. The explicit barrier after the parallel computation loop ensures all threads complete before sequential output processing begins.

4.6 Summary and Motivation

This chapter introduced a CPU parallelization approach for the Davidson eigenvalue solver utilizing k-point batching. The method leverages the inherent independence of electronic structure calculations across different k-points, facilitating embarrassingly parallel execution via OpenMP threading. Each thread processes its designated k-points autonomously, eliminating the need for inter-thread communication during computation. Although the k-point batching approach effectively accelerates Davidson diagonalization on multi-core CPUs, further performance gains are limited by CPU computational capacity and memory bandwidth. Furthermore, the principal computational operations in Davidson, namely matrix-matrix multiplications and three-dimensional FFTs, are particularly amenable to GPU acceleration due to their high arithmetic intensity and structured memory access patterns. The following chapters investigate GPU acceleration of the Davidson solver. Chapter 5 introduces the second strategy, which focuses on enabling GPU-level concurrency through asynchronous stream-based execution. Building on the OpenMP-based batching framework developed in this chapter, each concurrently processed k-point is assigned to an independent CUDA stream via the asynchronous queue mechanism provided by OpenACC. The chapter describes the correspondence

between OpenACC asynchronous queues and native CUDA streams, the management of GPU library handles on a per-batch basis for cuBLAS, cuSOLVER, and cuFFT, and the multi-level synchronization approach designed to ensure computational correctness while maximizing concurrent utilization of the graphics processing unit.

5 GPU Asynchronous Stream Execution

5.1 The GPU Underutilization Problem

Chapter 4 introduced the first strategy of this thesis: CPU-level k-point batching implemented with OpenMP threading. This approach achieved near-linear performance improvements on multicore processors by leveraging the inherent computational independence of the Brillouin zone sampling points. In addition to providing performance gains on the host side, this strategy established the essential structural framework required for GPU acceleration. Specifically, it introduced concurrent processing of k-points, batched data layouts, and thread-private identifiers for execution streams, all of which form the basis for the GPU methodology presented in this chapter.

The second strategy, described here, extends this concurrency model to GPUs by enabling asynchronous stream-based execution with OpenACC. This approach directly addresses the central challenge highlighted in this thesis, namely, the underutilization of GPUs in small-scale Quantum ESPRESSO calculations. Although GPUs provide massive parallelism through thousands of hardware threads, simply porting the single k-point Davidson solver to the device does not guarantee high utilization.

A single Davidson iteration for one k-point consists of a sequence of relatively short operations, including dense matrix multiplications, subspace diagonalization, preconditioning, and normalization. Each of these tasks occupies the GPU only briefly before control returns to the host, resulting in idle time while subsequent work is prepared. Because these operations are too small to fully engage the large number of available GPU cores, the device spends a substantial portion of the total execution time waiting rather than performing useful computation. This behavior characterizes what is referred to in this thesis as small-size jobs, in which the workload associated with each k-point is insufficient to fully utilize the device. Such conditions commonly arise in practical materials science simulations with limited k-point counts, moderate plane-wave cutoffs, or relatively small unit cells.

A critical observation underpinning the proposed solution is that k-points are mathematically independent. While they share global system parameters such as the Hamiltonian form, pseudopotential definitions, and FFT grids, their wavefunctions, eigenvalues, and convergence are entirely decoupled. Consequently, no data dependencies exist between Davidson iterations corresponding to different k-points. This property enables concurrent execution of multiple solvers on separate CUDA streams. The objective of this chapter is therefore to present an implementation that exploits this independence to increase GPU utilization by executing multiple Davidson solvers simultaneously, each assigned to a distinct CUDA execution stream.

5.2 OpenACC as the GPU Programming Model

QE is an extensive Fortran based software suite maintained by a global research network. While initial GPU integration began with CUDA, OpenACC is currently the preferred framework for expanding GPU support. This choice is driven by the efficiency of directive based models, which minimize architectural changes and simplify maintenance by keeping the codebase unified. Furthermore, OpenACC allows for seamless interoperability with existing CUDA code where specialized performance is still required. These compiler annotations enable selected regions of existing Fortran source code to be offloaded to the GPU, with the compiler automatically generating the corresponding device code. This approach preserves a unified codebase that supports both CPU-only and GPU-accelerated execution paths, controlled through conditional compilation directives present throughout the solver implementation.

5.2.1 OpenACC Directives

The !*$acc enter data create* directives allocate device-side memory without transferring any values from the host. It is used at the start of cegterg for the working arrays `psi`, `hpsi`, and `spsi`, and in the driver program for the persistent batched arrays `evc_batched`, `eig_batched`, `fft_array_batched`, and `aux_batched`. The corresponding !`$acc exit data delete` directives free the device memory at the end of the subroutine.

The !`$acc declare device_resident` directive, applied to `hc`, `sc`, and `vc` in `cegterg.f90`, tells the compiler that these arrays live permanently on the GPU from the moment they are allocated. They are never automatically synchronized to the host. This is correct for the reduced Hamiltonian, overlap, and eigenvector matrices, which are constructed, updated, diagonalized, and discarded entirely on the device across all the Davidson iterations.
The `!$acc` host_data use_device directive bridges the OpenACC memory model and external CUDA library calls by temporarily substituting device pointer values in place of host pointer variables. This interoperability mechanism is essential: without it, passing Fortran array names to `cuBLAS` routines like `ZGEMM` would provide host addresses rather than the required GPU addresses.
Compute kernels are offloaded with !`$acc parallel loop` for explicitly parallel loops and !`$acc kernels` for regions where the compiler infers parallelism. The *collapse (N)* clause merges N nested loops into a single index space, maximizing the number of concurrent GPU threads.

5.2.2 Central role of Async Clause

Every OpenACC directive within the cegterg routine includes the async clause with an identifier parameter, where this identifier is assigned at the subroutine entry from the thread private variable clock_thread introduced in Section 5.3. This clause ensures that all the GPU operations associated with a given k-point are submitted to a single

ordered asynchronous queue rather than being executed synchronously. When an OpenACC directive specifies asynchronous execution, the host thread immediately resumes execution after dispatching the operation, without waiting for completion on the device. The CPU can therefore proceed to enqueue additional work or return control to the OpenMP scheduler. As a result, the entire Davidson iteration corresponding to a single k-point, from the initial memory allocation to the final eigenvector updates, is represented on the GPU as an ordered sequence of queued operations within a single asynchronous execution stream. These operations are executed in the order in which they were submitted, while the host processor remains free from blocking. Synchronization between the host and the device occurs only once per batch of k-points at an explicit barrier, as described in Section 5.4.

5.3 CUDA Streams and the Asynchronous Execution Model

5.3.1 Concept of a CUDA Stream

A CUDA stream is an ordered sequence of GPU operations, including memory transfers, kernel launches, and library function calls. Operations within a single stream are executed strictly in submission order, whereas operations belonging to different streams are independent and may be overlapped by the hardware scheduler when sufficient resources are available [NVI24b].
This execution model aligns naturally with the k-point batching strategy. By assigning each k-point to an independent stream, the GPU can execute operations for different k-points concurrently. For example, matrix operations for one k-point may proceed while subspace diagonalization is still ongoing for another. This overlap reduces idle periods and significantly increases the overall device utilization.

5.3.2 Relationship Between OpenACC Queues and CUDA Streams

OpenACC defines asynchronous execution queues, identified by integer indices. Each queue corresponds internally to a native CUDA stream managed by the runtime system. The mapping between these abstractions can be obtained through a runtime query that returns the CUDA stream associated with a given OpenACC queue identifier. The key function that exposes this mapping is:

```
1 clock_cuda_stream = acc_get_cuda_stream(clock_thread)
```

Once this mapping is established, the resulting CUDA stream handle can be passed directly to GPU library routines such as cuBLAS, cuSOLVER, and cuFFT. This mechanism provides the essential connection between the directive-based OpenACC model and the native CUDA library ecosystem in such a multistream framework. By binding library operations to the same stream identifier used by OpenACC directives, all computations related to a specific k-point are executed within a single ordered stream, thereby preserving correct execution sequencing without requiring explicit synchronization.

5.3.3 Thread Private Stream Identification

A correct stream assignment in a multithreaded environment requires that each OpenMP thread consistently uses a distinct asynchronous queue identifier. This is achieved by thread-private variables that store both the queue identifier and the corresponding CUDA stream handle. Each thread initializes these variables at the beginning of its execution and subsequently uses them throughout the solver call hierarchy:

```
1   !$omp do
2  do i_batch = 1, min(nk_batches, nks - ik + 1)
3           clock_thread       = i_batch
4     clock_cuda_stream   = acc_get_cuda_stream(clock_thread)
5     ...
```

When cegterg is subsequently called from within this parallel region, it reads clock_thread and clock_cuda_stream from the calling thread's private copies:

```
1   async_id        = clock_thread
2  mycudaStream    = clock_cuda_stream
```

Because these variables are private to each thread, operations issued by different threads are automatically routed to separate execution streams. This design avoids modifications to the Davidson solver interface, since stream identity propagates implicitly through thread-private global variables—module-level variables declared with !$omp threadprivate—rather than through explicit function arguments.

5.3.4 Asynchronous Memory Transfers

Memory transfer operations within the solver are implemented using asynchronous device memory routines that enqueue data movement requests on the CUDA stream associated with the corresponding k-point. These operations use two-dimensional transfer functions to account for differences in leading dimensions across array storage. The asynchronous nature of these routines allows data movement to proceed concurrently with computation, enabling the host processor to schedule subsequent tasks without waiting for transfer completion.

5.4 Synchronization Strategy

The implementation employs three levels of synchronization to ensure correct execution ordering while preserving concurrency.

5.4.1 Ordering Within a Stream

Operations issued to a single CUDA stream are executed strictly in submission order, which inherently guarantees the correct sequencing of all steps within the Davidson

algorithm for a given k-point. The async*(ASYNC_ID)* clause on every directive ensures they are all enqueued to the same stream, so the stream's ordering guarantee applies to the entire Davidson iteration without any additional synchronization calls. Consequently, no explicit synchronization is required at this level.

5.4.2 Synchronization Across Streams

After the completion of the OpenMP parallel region, the host threads may have finished execution while the GPU streams continue processing queued operations. If the host were to read eig_batched immediately, it would see incomplete eigenvalues. The two-step synchronization in the device addresses this:

```
1 !$omp end parallel
2 call stop_clock('davidson')
3 !$acc wait
4 !$acc update self(eig_batched)
```

The !$acc wait directive without an argument waits for all pending operations across all the async queues. A global device synchronization barrier is therefore required to ensure that all the pending GPU work is completed before results are accessed on the host. This barrier guarantees data consistency when transferring eigenvalues back to the host memory.

5.4.3 Host and Device Consistency for Convergence Checks

Within the Davidson iteration, the convergence array *conv* computed on the device must be available on the host for termination checks. This is handled by the copy clause on the convergence-testing kernel:

```
1 !$acc parallel loop copy(conv(1:nvec)) copyin(btype(1:nvec))
2       async(async_id)
3 DO i = 1, nvec
4    IF( btype(i) == 1 ) THEN
5       conv(i) = ( ABS( ew(i) - e(i) ) < ethr )
6    ELSE
7       conv(i) = ( ABS( ew(i) - e(i) ) < empty_ethr )
8    END IF
9 END DO
```

The *copy* clause performs an upload of *conv* to the device before the kernel execution and a download to the host after it. Because the directive also carries *async (async_id)*, the download is enqueued on the stream rather than executed synchronously. At the point where COUNT *(.NOT. conv(:))* is called on the host, the host implicitly waits for the preceding async operations to complete, ensuring the host sees the correct convergence

state. This implicit synchronization is the only host-device synchronization that occurs inside the Davidson loop; all other operations proceed asynchronously.

5.4.4 MPI Communication and Device Pointers

The intra-band-group MPI reductions—collective operations among the subset of MPI processes that share responsibility for a group of eigenvectors—accumulate the reduced Hamiltonian and overlap matrix contributions using a CUDA-aware MPI path under the _CUDA preprocessor guard.:

```
#if defined(__CUDA)
CALL mp_sum( hc, 1, nbase, n_start, n_end, intra_bgrp_comm )
#else
CALL mp_sum( hc(1:nbase, n_start:n_end), intra_bgrp_comm )
#endif
```

These operations act directly on device memory pointers, avoiding unnecessary data transfers and ensuring consistency within the asynchronous execution framework.

5.5 Management of GPU Library Handles

Modern GPU computing relies on several specialized libraries, each providing optimized implementations of common operations: cuBLAS for dense linear algebra, cuSOLVER for eigenvalue problems, and cuFFT for Fast Fourier Transforms. Each library uses a handle object that encapsulates the execution context, including the associated CUDA stream. To maintain concurrency across k-points, independent handles are allocated for each batch.

5.5.1 cuBLAS Handle Management

Matrix multiplications within the Davidson algorithm are performed via cuBLAS using batch-specific handles bound to the corresponding CUDA stream [NVI24a]. This ensures that operations for different k-points are executed concurrently without serialization. A per-batch cuBLAS handle is created at the entry of cegterg and destroyed at the exit:

```
type(cublasHandle) :: myblasHandle(20)
...
istat_cublas = cublasCreate(myblasHandle(i_batch))
istat_cublas = cublasSetStream(myblasHandle(i_batch), mycudaStream)
...
! [Davidson iterations]
...
istat_cublas = cublasDestroy(myblasHandle(i_batch))
```

The cublasSetStream call binds handle myblasHandle(i_batch) to mycudaStream, which is the CUDA stream associated with k-point i_batch. All the subsequent ZGEMM calls, issued through this handle within the !acc host_data use_device region, therefore are executed on the correct stream and are correctly ordered relative to the OpenACC kernels on the same stream. The handle array is declared with a static size of 20 elements (sufficient for the batch sizes tested here; a production implementation could allocate dynamically based on `nk_batches`. Each concurrent invocation of cegterg accesses a different index of this array, so no lock or atomic operation is needed.

An alternative design would have been to allocate persistent cuBLAS handles at program start and pass them as arguments into cegterg [NVI24c]. The chosen approach of creating and destroying handles within the subroutine is simpler and avoids adding a handle argument to the interface. The overhead of cublasCreate and cublasDestroy is negligible relative to the Davidson iteration time. With nk_batches=4, the total handle management overhead is $\approx$ 2 ms, incurred once during initialization. Compared to the Davidson iteration time of $\approx$ 1,320 ms (from Nsight Systems profiling), this represents less than 0.15% overhead and is amortized across all SCF iterations.

5.5.2 cuSOLVER Handle via laxlib

Subspace diagonalization routines internally invoke cuSOLVER through the LAXlib interface. Multiple independent cuSOLVER contexts are created during initialisation and associated with individual streams, enabling simultaneous diagonalizations without requiring mutual exclusion mechanisms. The mechanism for associating the correct stream with cuSOLVER operations, therefore, requires a different approach that operates at the library initialization interface level. At the program startup, before the k-point loop begins initialize_cusolver_handles(nk_batches) This routine, part of the laxlib_cusolver_handles module, pre-allocates nk_batches independent cuSOLVER handles, numbered 1 through nk_batches. Each handle is a fully independent cuSOLVER context with its own internal workspace and stream association. Immediately after the stream initialization in the preparatory parallel region, each thread registers its stream with the corresponding handle:

```
1  !$omp do
2  do i_batch = 1, nk_batches
3     clock_thread       = i_batch
4     clock_cuda_stream  = acc_get_cuda_stream(clock_thread)
5     call initialize_laxlib_cuda_stream(clock_cuda_stream, clock_thread)
6     ...
```

The call *initialize_laxlib_cuda_stream(clock_cuda_stream, clock_thread)* binds the cuSOLVER handle number *clock_thread to clock_cuda_stream.* When diaghg is subsequently called from within *cegterg* on thread *i_batch*, it selects the cuSOLVER handle number *clock_thread* (which equals *i_batch*) and all its operations are executed on the stream associated with that batch slot.

This design resolves the contention issue that necessitated the OMP lock around *diaghg* in earlier iterations of the implementation. In the original code, a single shared cuSOLVER handle meant that concurrent calls from different threads would attempt to use the same handle simultaneously, causing incorrect results or crashes. The lock serialized these calls, thereby eliminating the concurrency benefit. With per-batch handles, the calls are genuinely independent, and the lock is no longer required, as evidenced by its commented-out state in the current source:

```
!call omp_set_lock(cegterg_locker)
IF( my_bgrp_id == root_bgrp_id ) THEN
   CALL diaghg( nbase, nvec, hc, sc, nvecx, ew, vc, ... )
END IF
!call omp_unset_lock(cegterg_locker)
```

This lock variable is preserved in the source for documentation purposes, recording the problem that existed and confirming that the current design has made it obsolete.

5.5.3 FFT Operations and the fftw_locker

Fast Fourier Transforms are used to apply the local potential in h_psi_ptr: the wavefunction is transformed from reciprocal space to real space, multiplied by the local potential, and then transformed back. The FFT arrays *fft_array_batched* and *aux_batched* carry the extra batch dimension *nk_batches* so that each concurrent k-point has its own workspace:

```
allocate( fft_array_batched(dfft%nnr, nk_batches) )
allocate( aux_batched(dfft%nnr, nk_batches) )
!$acc enter data create(fft_array_batched, aux_batched)
```

The fftw_locker initialized at program start guards the FFT plan creation. In FFTW-based CPU builds, the plan creation involves a global state that is not thread-safe; the lock serializes the plan creation while allowing for a concurrent plan execution. On the GPU path with cuFFT, an analogous concern is plan initialization: cuFFT plan handles, like cuBLAS and cuSOLVER handles, must be associated with the correct stream before use. The lock protects the initialization phase, while the per-batch workspace arrays ensure that the concurrent execution of FFT operations proceeds without interference.

5.6 Profiling and GPU utilization Results

To quantify the performance improvement achieved by the asynchronous stream-based batching strategy, Nsight Systems profiling was performed on the Leonardo supercomputer using the silicon test case (`si2_11_points.in`, 11 k-points) under three configurations: sequential execution (`nk_batches=1`), four-way concurrent execution (`nk_batches=4`), and fully concurrent execution processing all k-points in a single

batch (nk_batches=11). All the runs used the same binary compiled with NVHPC 24.5, OpenACC, and NVTX instrumentation enabled, executed on a single NVIDIA A100-SXM 64 GB GPU on the Leonardo supercomputer [TAC23].

5.6.1 Wall Time and Speedup

Table 5.1 summarizes the measured wall times, GPU kernel times and derived performance metrics for all the three configurations. The nk_batches=4 configuration achieves the

Configuration	Wall Time	GPU Kernel Time	GPU Utilization	Speedup vs nk1
nk_batches=1 (baseline)	3.60 s	~914 ms	~25%	1.00×
nk_batches=4	2.56 s	~979 ms	~38%	1.41×
nk_batches=11	3.40 s	~979 ms	~29%	1.06×

Table 5.1: Wall time and GPU utilization for sequential versus batched k-point execution on the 11 k-point silicon test case. The GPU kernel time is derived from the cuda_gpu_kern_sum report; The GPU utilization is the ratio of total kernel time to wall time.

best wall-time speedup of 1.41× over sequential execution. The total GPU kernel time remains nearly constant across all the configurations (~914–979 ms), confirming that the amount of GPU work is the same in all cases. The speedup arises entirely from packing this fixed amount of work into a shorter wall time through concurrent stream execution. The nk_batches=11 configuration, despite processing all the 11 k-points in a single batch with no batch boundary barriers, does not improve over nk_batches=4 and nearly regresses to the baseline performance. The reason is identified in Section 5.6.3.

5.6.2 Timeline Analysis

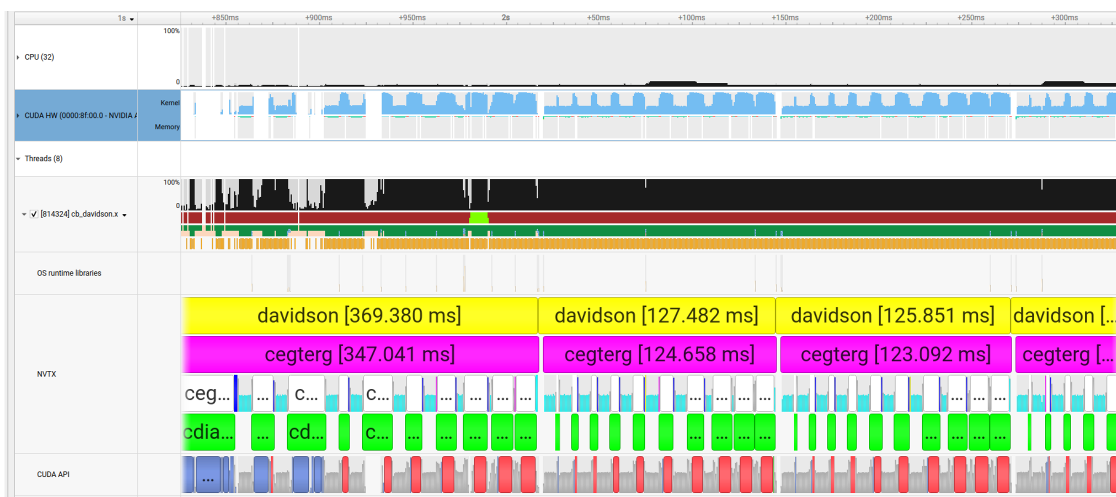

Figure 5.1: Nsight Systems timeline for nk_batches=1.

nk_batches=1 baseline figure 5.1: The Nsight Systems [NVI25] timeline for the sequential baseline shows a single active thread with Davidson and cegterg NVTX annotations appearing strictly one at a time [NVI24e]. Individual cegterg blocks measure approximately 124ms each. The CUDA HW Kernel row shows sparse activity, with visible idle gaps between successive Davidson solves — the GPU completes one k-point's worth of work and then sits idle while the host prepares the next. This sparse activity pattern is the defining characteristic of the small-size job problem addressed by this thesis.

Figure 5.2: Nsight Systems timeline for nk_batches=4.

nk_batches=4 concurrent execution figure 5.2: With four concurrent batch slots, the timeline shows four threads executing cegterg blocks simultaneously. Thread identifiers 814467, 814479, 814480, and 814481 all show active cegterg annotations at the same horizontal time position, confirming genuine concurrent execution on four independent CUDA streams. The CUDA HW Kernel row shows denser, more continuous activity than the baseline, with idle gaps significantly reduced. The wall time decreases from 3.60 s to 2.56 s as the GPU transitions from sequential one-at-a-time k-point processing to overlapping concurrent execution.

nk_batches=11 fully concurrent execution 5.3: Processing all 11 k-points in a single batch eliminates the batch boundary barriers entirely. The GPU Metrics rows—GR Active and SM Active, which respectively indicate the fraction of time the GPU has work scheduled and the fraction of time streaming multiprocessors are executing—show dense continuous activity throughout the computation. However, this elevated hardware activity does not correspond to productive computation: the ratio of GPU kernel execution time to wall time is only 29% at `nk_batches=11`, lower than the 38% achieved at `nk_batches=4`. The high GR Active metric reflects the GPU engine processing serialized lock overhead from `fftw_locker` contention rather than genuine concurrent k-point execution.

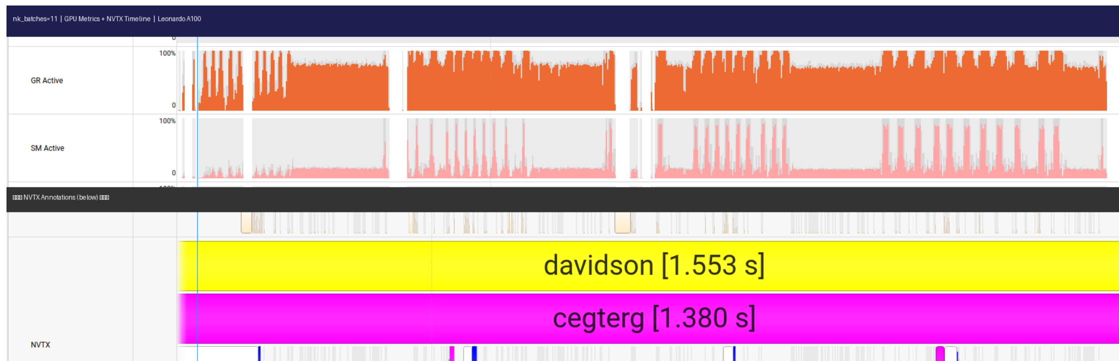

Figure 5.3: Nsight Systems GPU Metrics for nk_batches=11.

5.6.3 The FFT Lock Bottleneck

The failure of nk_batches=11 to achieve a proportional speedup reveals the remaining serialization bottleneck in the implementation: the fftw_locker OpenMP lock in my_h_psi_batched. This lock, introduced to protect the FFT plan initialization from concurrent thread access, serializes the invfft and fwfft calls across all the concurrent threads. With 11 threads all competing for this single lock, each thread must wait for every other thread's FFT sequence to complete before proceeding. The profiling timeline confirms this: instead of many short sequential cegterg blocks as in nk_batches=1, the nk_batches=11 timeline shows a single long cegterg block of duration 1.380 s per thread — the individual Davidson solution take far longer because threads spend most of their time waiting for the lock.

The CUDA GPU kernel summary quantitatively confirms the symptom. The most direct quantitative evidence for `fftw_locker` serialization comes from the NVTX `fftw` timing, which measures the total time each thread spends in the FFT region including lock waiting time. The average `fftw` duration increases from 24 μs at `nk_batches=1` to 76 μs at `nk_batches=4` and 1,301 μs at `nk_batches=11` — a 54× increase — confirming that threads spend the vast majority of their FFT time waiting for lock acquisition rather than executing FFT operations. The individual GPU kernel durations for `regular_fft` (4.0 μs at nk=1, 4.8 μs at nk=4 and nk=11) and `vector_fft` (3.4 μs at nk=1, 4.0 μs at nk=4, 4.2 μs at nk=11) show only modest increases, confirming that the kernels themselves execute efficiently once launched — the bottleneck is exclusively in the lock waiting time before kernel launch, not in the kernel execution itself.

This result directly motivates the future work described in Section 5.7: eliminating the FFT lock by pre-initializing FFT plans before the parallel region begins, so that the plan creation overhead does not require serialization during the Davidson iterations.

5.6.4 Kernel Distribution Analysis

Table 5.2 shows the dominant GPU kernels by execution time, which are consistent across all the configurations since the same operations are executed regardless of batching — only their temporal overlap changes.

The dominance of sytrd4_gpu at 33% of all the GPU kernel time confirms the theoretical

Kernel	% GPU	time Operation
sytrd4_gpu (two variants)	33%	cuSOLVER subspace diagonalization
. Tommaso Gorni. regular_fft + vector_fft	14%	FFT for local potential application in H|ψ⟩
ampere_zgemm + cutlass ZGEMM variants	8%	cuBLAS subspace projection
my_h_psi_batched_* kernels	7%	Custom H|ψ⟩ application kernels
cegterg_*_gpu kernels	3%	Davidson convergence and update operations

Table 5.2: Dominant GPU kernels in Davidson diagonalization, consistent across all the nk_batches configurations.

prediction in Section 5.7.1 that the cuSOLVER subspace diagonalization is the primary per-stream GPU bottleneck. This kernel implements the tridiagonal reduction step of the symmetric eigenvalue solver and scales as $O(N^3)$ with respect to the subspace dimension. For the subspace dimensions encountered in this test case (nvecx 200–400), the kernel is too small to fully saturate an A100's 6912 CUDA cores, limiting the utilization achievable within any single stream regardless of how many streams run concurrently.

5.6.5 Discussion

The profiling results demonstrate that the asynchronous stream-based batching strategy successfully addresses the GPU underutilization problem described in Section 5.1, achieving a 1.41× wall-time speedup with nk_batches=4 and raising the GPU utilization from 25% to 38%. The result falls short of the theoretical maximum for two well-understood reasons. First, the cuSOLVER diagonalization kernels consume 33% of thhe GPU time with limited internal parallelism, placing a ceiling on the per-stream utilization independently of concurrency. Second, the fftw_locker serializes FFT operations across concurrent threads, preventing the full benefit of stream overlap from materializing at higher batch counts.
Taken together, these results define a clear optimization roadmap. The immediate next step — pre-initializing FFT plans before the parallel region to eliminate the lock — is expected to recover the performance lost between nk_batches=4 and nk_batches=11, enabling all the 11 k-points to be processed concurrently without serialization. Beyond this, replacing cuSOLVER with a batched eigenvalue solver capable of processing multiple small problems simultaneously would address the remaining 33% bottleneck and substantially increase the utilization.

5.7 Summary

This chapter has described the design, implementation, and performance characterization of GPU-accelerated k-point batching for the Davidson eigenvalue solver in Quantum ESPRESSO. The central problem addressed is GPU underutilization in small jobs: a

single-k-point Davidson solver produces short bursts of GPU activity separated by idle intervals, because individual operations are too small to saturate modern GPU hardware. Profiling confirmed baseline utilization of approximately 25% for the 11 k-point silicon test case on Leonardo's A100 GPUs.

The solution exploits the mathematical independence of k-points established in Chapter 3 to run multiple Davidson solvers concurrently on the GPU. The implementation rests on four interconnected mechanisms. OpenACC provides the GPU offloading model, with async(async_id) ensuring all operations for a given k-point are enqueued to a single ordered queue without blocking the host CPU. The acc_get_cuda_stream function bridges the OpenACC's queue abstraction to native CUDA stream handles, enabling the CUDA library calls to be executed on the same stream as the OpenACC kernels. The Thread-private variables clock_thread and clock_cuda_stream propagate the stream identity through the call tree from the OpenMP parallel region down through cegterg without modifying the subroutine interfaces. Per-batch library handles — cuBLAS handles created within cegterg and cuSOLVER handles pre-allocated by initialize_cusolver_handles — eliminate the inter-thread serialization that a single shared handle would impose.

Performance profiling across three batch configurations quantified both the achievable gains and the remaining bottlenecks. The nk_batches=4 configuration achieved the best wall-time speedup of 1.41× with four concurrent streams visible in the Nsight Systems timeline. The nk_batches=11 configuration revealed that the fftw_locker serializes FFT operations across all the 11 threads, preventing proportional scaling despite excellent hardware-level GPU utilization as shown by the dense GR Active and SM Active metrics. Kernel distribution analysis identified cuSOLVER's sytrd4_gpu as consuming 33% of all the GPU kernel time, confirming the subspace diagonalization bottleneck predicted in Section 5.7.1.

The two strategies developed in Chapters 4 and 5 — OpenMP k-point batching and OpenACC asynchronous stream execution — constitute together, a practical combined approach for the GPU-optimized execution of small-size jobs in Quantum ESPRESSO, demonstrating that the underutilization problem is addressable within the existing Fortran codebase without algorithmic changes to the Davidson method itself. The remaining challenges — eliminating the FFT lock, addressing the cuSOLVER bottleneck, and integrating them to the full PWscf SCF loop — define a concrete roadmap for extending this proof-of-concept miniapp to production use.

6 Conclusions

6.1 Summary of Contributions

This thesis addressed the problem of GPU underutilization in small-sized electronic structure calculations performed with Quantum ESPRESSO. The central observation motivating this work is structural: when a calculation involves moderate plane-wave cutoffs or small unit cells, the Davidson eigenvalue solver at a single k-point does not provide sufficient computational work to saturate a modern GPU, and the large number of k-points generally involved for this type of calculation tends to aggravate the offloading overhead. The device executes a short burst of computation and then sits idle while the host prepares the next task. Profiling confirmed this behaviour concretely, measuring a baseline GPU utilization of approximately 25% for an 11 k-point silicon test case on Leonardo's NVIDIA A100-SXM GPUs.

Two complementary strategies were developed and validated within the `cb_davidson` miniapp, a standalone implementation that extracts the Davidson solver and its essential supporting routines from the full Quantum ESPRESSO codebase.

The first strategy, presented in Chapter 4, exploits the mathematical independence of k-points established in Chapter 3 to process multiple Davidson solution concurrently on the host CPU through OpenMP threading. A two-level loop structure processes k-points in configurable batches, with pre-allocated three-dimensional workspace arrays, ensuring thread-safe memory access without the overhead of dynamic allocation. A significant technical challenge encountered during the implementation — race conditions in the UtilXlib timing infrastructure caused by concurrent access to the module-level arrays – were diagnosed and resolved using a per-thread clock indexing architecture. Performance measurements on the DGX Spark workstation demonstrated speedups of 1.80× with 2 threads and 3.34× with 4 threads, achieving parallel efficiencies of 90% and 84% respectively. Beyond its direct CPU-level benefit, This batching framework established the structural foundation required for GPU concurrency: concurrent k-point processing, batched data layouts, and thread-private stream identifiers.

The second strategy, presented in Chapter 5, extended this concurrency model to the GPU through asynchronous stream-based execution using OpenACC. Each concurrent k-point is assigned an independent CUDA stream via the OpenACC `async` clause, with stream handles retrieved through `acc_get_cuda_stream` and propagated to the cuBLAS, cuSOLVER, and cuFFT library calls through thread-private variables. Per-batch cuBLAS handles created within `cegterg` and pre-allocated cuSOLVER handles managed through the LAXlib interface ensure that all the GPU operations for a given k-point are executed on a single ordered stream without inter-k-point serialization. Profiling with NVIDIA Nsight Systems and NVTX instrumentation across three configurations quantified the achieved improvement: `nk_batches=4` delivered a wall-time speedup of 1.41× and raised GPU utilization from 25% to 38%. The `nk_batches=11` configuration revealed the `fftw_locker` as the remaining serialization bottleneck. The GPU Metrics

rows showed high GR Active utilization while the NVTX timeline simultaneously showed a single long `cegterg` block of 1.380 s rather than 11 short overlapping blocks, confirming that the GPU was kept busy by serialized stream operations queuing behind the `fftw_locker` rather than by genuine concurrent k-point execution.

Kernel distribution analysis identified `sytrd4_gpu` — the cuSOLVER tridiagonal reduction kernel — as consuming 33% of all the GPU kernel time, confirming the subspace diagonalization is the primary per-stream bottleneck. For the subspace dimensions encountered in this test case ($N_{vecx} \sim 200$–400), this kernel is too small to fully saturate the A100's 6912 CUDA cores, placing an upper bound on the per-stream utilization that is independent of how many streams are executed concurrently.

Together, the two strategies demonstrate that the GPU underutilization in small-sized QE jobs are addressable within the existing Fortran codebase without algorithmic changes to the Davidson method itself.

6.2 Future Developments

The profiling results presented in this thesis define a concrete roadmap for extending the proof-of-concept implementation toward production-ready performance.

The strategies validated in `cb_davidson` are directly transferable to the production Quantum ESPRESSO codebase, but require non-trivial refactoring of the full `h_psi` routine. The production implementation relies on globally shared workspace arrays without a batch dimension and assumes single k-point execution throughout. Making it thread-safe for concurrent k-point processing requires adding the batch index dimension to these shared arrays, which propagates changes across multiple interdependent source files, including `h_psi.f90`, `s_psi.f90`, `g_psi.f90`, and the Hamiltonian assembly routines.

Beyond this refactoring effort, full integration would also enable two additional optimizations not explored in this thesis: overlapping MPI communications with GPU computations across k-points, and extending the batching strategy to the outer SCF loop to amortize the initialization overhead across multiple self-consistent iterations. These extensions would be particularly beneficial for production calculations involving moderate system sizes, where both k-point count and individual operation sizes are insufficient to fully utilize the GPU independently.

A further opportunity for performance improvement lies in batching the subspace diagonalization step across concurrent k-points. In the current implementation, each k-point stream calls the cuSOLVER independently to solve its own small subspace eigenvalue problem. Rather than solving these problems independently per stream, the subspace eigenvalue problems from all concurrent k-points could be collected and submitted as a single batched operation.

6.3 Closing Remarks

The results of this thesis demonstrate that combining OpenMP k-point batching with OpenACC asynchronous stream execution is an effective strategy for improving GPU utilization in small-sized QE calculations. The 1.41× wall-time speedup achieved on Leonardo's A100 GPUs, together with the clear identification of the two remaining bottlenecks and their proposed solutions, provides a practical foundation for further development. We expect the speedup to increase substantially when using a more realistic and computationally expensive `h_psi`, in which, for example, a full-weight non-local potential operator must be applied.

The miniapp methodology proved effective for validating these strategies in a controlled environment: the same Davidson solver and the library stack used in production Quantum ESPRESSO confirmed that the techniques are directly applicable to the full codebase.

As HPC systems increasingly rely on GPU accelerators as their primary source of computational throughput, ensuring that electronic structure codes can efficiently utilize these devices for the full range of calculation sizes — not only large systems, but also small ones with many k-points — becomes increasingly important. The strategies developed here contributes a concrete step toward this goal for QUANTUM ESPRESSO's Davidson eigensolver.

Bibliography

[BO27] M. Born and J. R. Oppenheimer. **Zur Quantentheorie der Molekeln**. *Annalen der Physik* 389:20 (1927), 457–484. DOI: 10.1002/andp.19273892002 (see page 5).

[Dai+14] Xiaoying Dai, Xingao Gong, Aihui Zhou, and Jinchao Zhu. **A parallel orbital-updating approach for electronic structure calculations**. *arXiv preprint arXiv:1405.0260* (2014) (see page 9).

[Dai+15] Xiaoying Dai, Zhaojun Liu, Xuejiang Zhang, and Aihui Zhou. **A parallel orbital-updating based optimization method for electronic structure calculations**. *arXiv preprint arXiv:1510.07230* (2015) (see page 9).

[Dav75] Ernest R. Davidson. **The iterative calculation of a few of the lowest eigenvalues and corresponding eigenvectors of large real-symmetric matrices**. *Journal of Computational Physics* 17:1 (1975), 87–94 (see page 9).

[Don+90] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain S. Duff. **A set of level 3 basic linear algebra subprograms**. *ACM Transactions on Mathematical Software (TOMS)* 16:1 (1990), 1–17. DOI: 10.1145/77626.79170 (see page 18).

[Gia+09] Paolo Giannozzi, Stefano Baroni, Nicola Bonini, Matteo Calandra, Roberto Car, Carlo Cavazzoni, Davide Ceresoli, Guido L Chiarotti, Matteo Cococcioni, Ismaila Dabo, et al. **QUANTUM ESPRESSO: a modular and open-source software project for quantum simulations of materials**. *Journal of Physics: Condensed Matter* 21:39 (2009), 395502. DOI: 10.1088/0953-8984/21/39/395502 (see page 24).

[Gia+17] P. Giannozzi, O. Andreussi, T. Brumme, O. Bunau, M. Buongiorno Nardelli, M. Calandra, R. Car, C. Cavazzoni, D. Ceresoli, M. Cococcioni, N. Colonna, I. Carnimeo, A. Dal Corso, S. de Gironcoli, P. Delugas, R. A. DiStasio Jr., A. Ferretti, A. Floris, G. Fratesi, G. Fugallo, R. Gebauer, U. Gerstmann, F. Giustino, T. Gorni, J. Jia, M. Kawamura, H-Y Ko, A. Kokalj, E. Kucukbenli, M. Lazzeri, M. Marsili, N. Marzari, F. Mauri, N. L. Nguyen, H-V Nguyen, A. Otero-de-la Roza, L. Paulatto, S. Ponce, D. Rocca, R. Sabatini, B. Santra, M. Schlipf, A. P. Seitsonen, A. Smogunov, I. Timrov, T. Thonhauser, P. Umari, N. Vast, X. Wu, and S. Baroni. **Advanced capabilities for materials modelling with Quantum ESPRESSO**. *Journal of Physics: Condensed Matter* 29:46 (2017), 465901. DOI: 10.1088/1361-648X/aa8f79 (see page 6).

[Gia+20] Paolo Giannozzi, Oscar Baseggio, Pietro Bona, 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 (Apr. 2020), 154105. DOI: 10.1063/5.0005082 (see page 6).

[HK64] P. Hohenberg and W. Kohn. **Inhomogeneous Electron Gas**. *Physical Review* 136 (1964), B864–B871 (see page 5).

[HSC79] D. R. Hamann, M. Schlüter, and C. Chiang. **Norm-Conserving Pseudopotentials**. *Physical Review Letters* 43:20 (1979), 1494–1497. DOI: 10.1103/PhysRevLett.43.1494 (see page 16).

[Kit05] Charles Kittel. **Introduction to Solid State Physics**. 8th. Hoboken, NJ: John Wiley & Sons, 2005 (see page 11).

[KS65] W. Kohn and L. J. Sham. **Self-Consistent Equations Including Exchange and Correlation Effects**. *Physical Review* 140 (1965), A1133–A1138 (see page 5).

[Mar20] Richard M. Martin. **Electronic Structure: Basic Theory and Practical Methods**. 2nd. Cambridge: Cambridge University Press, 2020 (see page 5).

[Nis24] Nisha. *KS_SOLVER_CB*. https://github.com/Nisha2592/cb_davidson/tree/main. GitHub Repository. 2024 (see page 25).

[NVI24a] NVIDIA Corporation. *cuBLAS Library User Guide*. NVIDIA Developer Documentation. 2024. URL: https://docs.nvidia.com/cuda/cublas/ (see page 42).

[NVI24b] NVIDIA Corporation. *CUDA C++ Programming Guide*. NVIDIA Developer Documentation. 2024. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide (see page 39).

[NVI24c] NVIDIA Corporation. *cuSOLVER Library User Guide*. NVIDIA Developer Documentation. 2024. URL: https://docs.nvidia.com/cuda/cusolver/ (see page 43).

[NVI24d] NVIDIA Corporation. *NVIDIA DGX Spark*. 2024. URL: https://www.nvidia.com/en-us/products/workstations/dgx-spark/ (see page 33).

[NVI24e] NVIDIA Corporation. *NVIDIA Tools Extension Library (NVTX)*. https://github.com/NVIDIA/NVTX. 2024 (see page 46).

[NVI25] NVIDIA Corporation. *Nsight Systems User Guide version 2025.1*. 2025. URL: https://docs.nvidia.com/nsight-systems/ (see page 46).

[Pay+92] M. C. Payne, M. P. Teter, D. C. Allan, T. A. Arias, and J. D. Joannopoulos. **Iterative minimization techniques for ab initio total-energy calculations: molecular dynamics and conjugate gradients**. *Reviews of Modern Physics* 64:4 (1992), 1045–1097. DOI: 10.1103/RevModPhys.64.1045 (see page 17).

[Pic89] Warren E. Pickett. **Pseudopotential methods in condensed matter applications**. *Computer Physics Reports* 9:3 (1989), 115–197. DOI: 10.1016/0167-7977(89)90002-6 (see pages 6, 16).

[TAC23] Matteo Turisini, Giorgio Amati, and Mirko Cestari. **Leonardo: A pan-european pre-exascale supercomputer for HPC and AI applications**. In: *Euro-HPC Conference*. 2023 (see page 45).

[Van90] David Vanderbilt. **Soft self-consistent pseudopotentials in a generalized eigenvalue formalism**. *Physical Review B* 41:11 (1990), 7892–7895. DOI: 10.1103/PhysRevB.41.7892 (see page 16).