A Massively Parallel Lockstep Pipeline For Full Isomerspace Optimisation

M.Sc. Thesis

by Jonas Dornonville de la Cour

Advisors
Assoc. Prof. James E. Avery
Prof. Markus Jochum
Carl-Johannes Johnson, Ph.D.

University of Copenhagen
Niels Bohr Institute

Abstract

Fullerenes are carbon molecules with a hollow cage-like structure. Their surfaces are made up exclusively of pentagon and hexagon rings. The number of theoretically stable fullerenes is infinite, growing with $\mathcal{O}(N^9)$, only a small handfull of which have been synthesised. Those that have, have shown great promise in a variety of uses, such as allergy and asthma medicine, cancer treatments, solar cells, biosensors, and printable electronics. Therefore, it is of great interest to be able to compute the properties of these molecules. Ideally we would like to be able to search through entire isomerspaces, to find those with the right properties.

However, it is completely infeasible to analyse more than a single fullerene using full ab-initio calculations, as even density functional theory (DFT) takes a week of compute for C_{60}. Thus, a more efficient approach is needed. Forcefield (FF) methods are far more computationally efficient, and have been shown to produce results that are of comparable quality to DFT optimised geometries. Current state-of-the-art FF methods are able to compute optimal geometries for C_{200} isomers in $100\mu s$. While this is certainly fast, it is not yet sufficient for full isomerspace exploration.

This thesis presents a fully lockstep parallel implementation of a pipeline for isomerspace forcefield optimisation, capable of leveraging thousands or even millions of compute cores. Our lockstep parallel approach attains roughly 3 orders of magnitude (950-1400x) faster performance than previous state-of-the-art FF implementation. Our implementation demonstrates essentially perfect scaling, consequently enabling further performance gains for larger isomerspaces, given sufficient compute resources.

The final pipeline allows us to exhaustively produce and optimise the entire C_{200} isomerspace ($2{\cdot}10^6$ isomers) in 6 hours on two GPUs, making what was previously completely impractical (247 days), possible within the span of an afternoon. This enables full isomerspace exploration, and sets the stage for molecular property analysis of billions of fullerenes.

Acknowledgements

I would like to thank James Avery for his understanding and support, without which this thesis would not have been possible. And for always encouraging me in my ideas and allowing me to pursue them. I would also like to thank Carl-Johannes Johnson for his instrumental supervision in the writing process, his calm and pragmatic advice and feedback was invaluable. I want to thank Markus Jochum for his patience and good will and ultimately for letting me pursue different goals than had originally been envisioned.

Lastly thank you to, all of my friends, who remained encouraging and supportive throughout the journey that this has been.

Contents

Contents 4

1 Introduction 7
1.1 Motivation . . . 8
1.2 Code Overview . . . 10

2 Graphs and Forcefield Methods 13
2.1 Topology of Fullerenes . . . 14
2.2 Forcefield Optimisation . . . 17
2.3 Summary . . . 22

3 Hardware 23
3.1 GPGPUs . . . 25
3.2 Memory . . . 34
3.3 Benchmarking Methodology . . . 36
3.4 Parallel Primitives . . . 38
3.5 Summary . . . 42

4 Design and Implementation of Lockstep Parallel Forcefield Algorithm 45
4.1 Overview and Motivation . . . 46
4.2 First Design . . . 48
4.3 Pipeline 1: CUDA/C++ Forcefield . . . 63
4.4 Validation and Convergence . . . 68
4.5 Pipeline 2: IsomerQueue Forcefield . . . 73
4.6 Performance . . . 78
4.7 Flatness . . . 84
4.8 Summary . . . 91

5 Fully Lockstep Parallel Pipeline 95
5.1 Pipeline 3: Tutte Embedding . . . 96
5.2 Pipeline 4: Spherical Projection . . . 106
5.3 Pipeline 5: Dualisation . . . 116
5.4 Pipeline 6: Pipeline Parallelism . . . 123
5.5 Pipeline 7: Multi-GPU Support . . . 124
5.6 Summary . . . 125

6 Conclusion and Future Work 129

6.1 Future Work 130
6.2 Conclusion 130

Bibliography 133

A System Specifications 135
A.1 Compilers 138

B Implementation Code 139

C Benchmark Scripts 149

Chapter 1
Introduction

1.1 Motivation

Fullerenes are carbon allotropes which form single atom thick polyhedral shells of carbon atoms. Their bond structures are cubic planar graphs made entirely of pentagons and hexagons. Their surfaces resemble graphene sheets and like these have extreme electron mobility and tensile strength. Unlike graphene, fullerenes come in many forms, each with their own electromagnetic, optical, thermodynamic, and mechanical molecular properties.

Fullerenes were first discovered in 1985 by Harold Kroto, Richard Smalley and Robert Curl. They were discovered by the use of a mass spectrometer, which was used to analyse the gas produced by the pyrolysis of graphite. The mass spectrometer was able to detect the presence of a new molecule, which was later identified as C_{60}, the first fullerene. The discovery of fullerenes was awarded the Nobel Prize in Chemistry in 1996. Since their discovery there has been a large amount of research into fullerenes, with many applications.

Only a few fullerenes have been produced, but these have found a wide variety of uses from asthma medicine, cancer treatments, solar cells, biosensors to printable electronics. Therefore, it is of great interest to be able to compute geometries and properties of these molecules.

It is, however, currently completely infeasible to analyse much more than a single fullerene using full ab-initio calculations like DFT. Therefore, we require new methods and theory to discover structures that can be synthesised and poses useful properties. In this thesis we want to build the technical foundation for exploration of *full* isomerspaces, comprising millions or even billions of distinct molecules, in order to find those few of particular interest, that warrant weeks of supercomputer time for full quantum chemical analysis.

Thurston established that the number of fullerene isomers for a given number of vertices N grows with $\mathcal{O}(N^9)$, e.g. there are 1812 C_{60} fullerene isomers and 214,127,742 C_{200} fullerene isomers. Figure Fig. 1.1 shows this scaling for the first 400 isomerspaces.

While forcefield optimisations are significantly faster than DFT or other quantum chemical methods, it still takes roughly $100ms$ to optimise a single C_{200} fullerene, so a back of the envelope calculation shows that it would take roughly 247 days to optimise all 214,127,742 C_{200} fullerenes. This realisation motivates the main research question of this thesis: *Can we use a lockstep parallel algorithm to exploit massively parallel hardware enabling exhaustive exploration of full isomerspaces?*

1.1.1 CARMA

The **CARMA** (**CAR**bon **MA**nifolds) project at the helm of James Avery, is a project which aims to take the development of fullerenes to the next level. The project aims to develop new mathematical and computational methods to analyse the properties of fullerenes

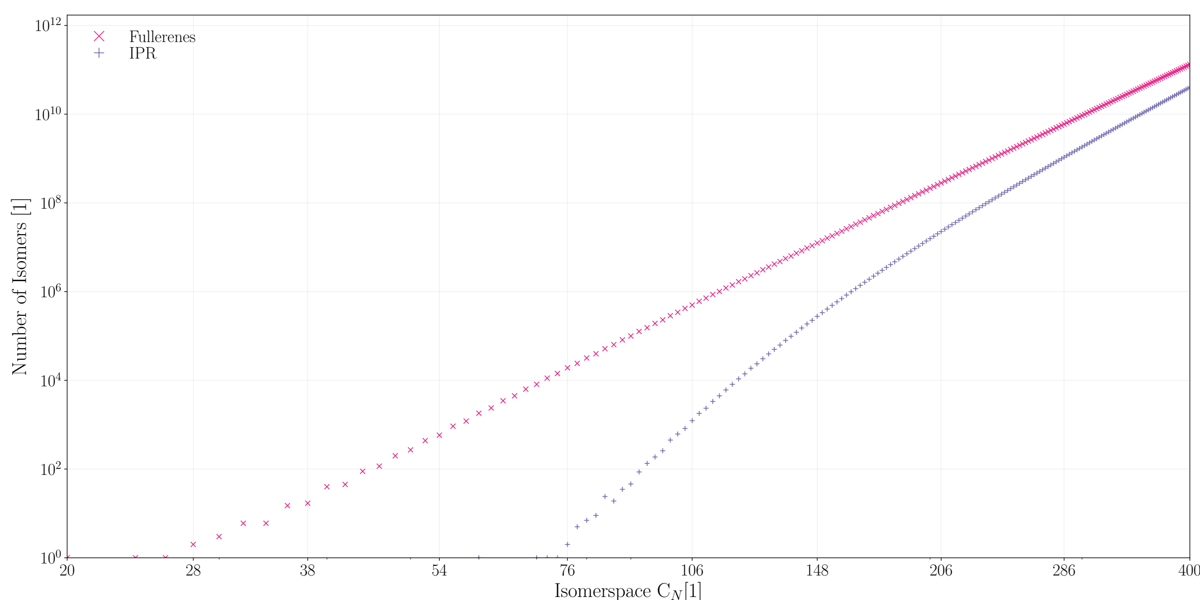


Figure 1.1: *Number of regular* x */* $+$ *IPR fullerenes for a given number of vertices* N*. Note the double logarithmic axes.*

orders of magnitude faster than current methods. The project consists 6 subprojects which are all aimed at different aspects of the project. The subprojects are as follows:

- **Task M**: Mathematics of Carbon Manifolds
- **Task W**: Wave Equations on Carbon Manifolds
- **Task E**: Isometric Embeddings
- **Task F**: Fulleroids, Schwarzites and other Carbon Manifolds
- **Task P**: Approximation of Molecular Properties
- **Task R**: Paths to Rational Synthesis

The primary focus of this thesis is **Task E**: *Isometric Embeddings* and further covers the groundwork necessary for fast approximations of certain molecular properties. Furthermore, the architecture of tools developed here could also be generalised to support other Carbon Manifolds such as Fulleroids (generalisation of fullerenes, allowing for heptagons, octagons et cetera).

The *Fullerene* program was created to be a versatile, open-source tool for generating and analysing fullerene isomers. It can produce symmetrical, two-dimensional drawings of fullerene graphs and generate precise three-dimensional molecular geometries using force-field optimisation. In addition, the program can perform topological and graph theoretical analysis and calculate various physical and chemical properties. It can serve as a useful starting point for more advanced quantum theoretical calculations. The program is written in standard C++ and Fortran and is available for download on GitHub at `https://github.com/jamesavery/fullerenes`.

1.2 Code Overview

For the present report, a significant amount of code had to be written, and we aim to give an overview of the code base found in the repository at `https://github.com/jamesavery/fullerenes/tree/development`

- `src/cuda` contains all the CUDA code written for the project.
  - `coord2d.cu` Contains the implementation of the `coord2d` type, a 2D coordinate structure used in the parallel lockstep Tutte embedding implementation.
  - `coord3d.cu` Contains the implementation of the `coord3d` type, a 3D coordinate structure used primarily in the parallel forcefield implementation.
  - `sym_mat_3.cu` Contains an efficient 3x3 symmetric matrix implementation, `SymMat3`. It is used in the parallel lockstep forcefield implementation, and allows for computation of eigenvalues and eigenvectors using a closed form solution with numerical stability guardrails.
  - `forcefield.cu` Contains the implementation of the parallel lockstep forcefield optimisation. see Section 4.2 for a detailed walkthrough of the algorithms.
  - `tutte.cu` Contains parallel lockstep Tutte embedding implementation, see Algorithm 21.
  - `dualize.cu` Contains the implementation of the parallel lockstep dualisation algorithm, see Algorithm 26.
  - `spherical_projection.cu` Contains the implementation of the semi-parallel spherical projection algorithm, see Algorithm 24.
  - `device_deque.cu` Contains the implementation of the `CuDeque` structure, a device side, grid-parallel (bot not inter-block parallel) implementation of a double ended queue, used in the MSSPs Algorithm 23.
  - `device_cubic_graph.cu` Contains an implementation of the cubic graph utility functions used in various algorithms, see Algorithm 7.
  - `device_dual_graph.cu` Contains an analogous implementation to the cubic graph, for triangulated graphs used in the dualisation algorithm.
  - `constants.cu` Contains the implementation of the Constants utility data structure, see Section 4.3.1 and Algorithm 8.
  - `node_neighbours.cu` Contains an implementation of the node neighbours utility data structure, see Section 4.3.1.
  - `reductions.cu` Contains an implementation of the reduction and scan methods used in various algorithms throughout, see Algorithm 4,Algorithm 3 and Algorithm 6.
  - `isomer_batch.cu` Contains the implementation of the `IsomerBatch` see Fig. 4.6.

  - `cuda_io.cu` Contains I/O functions that act on `IsomerBatch` structures, see Fig. 4.7.
  - `isomer_queue.cu` Contains the implementation of the `IsomerQueue` see Section 4.5.
  - `launch_dims.cu` Contains the implementation of the `LaunchDims` structure, which is only called from kernel launch sites, it computes the optimal launch configuration for a given kernel and jbo configuration and stores them statically in the function.
  - `cu_array.cu` Contains the implementation of the `CuArray` structure, which is a vector like structure implemented using CUDA Unified Memory, used to communicate arbitrary data between host and device.
  - `launch_ctx.cu` Contains the implementation of the `LaunchCtx` structure, which is to control the stream and device on which to launch a kernel or memory operation, from C++ code.

- `include/fullerenes/gpu` contains all the header files through which the necessary CUDA code is accessed from C++.
  - `kernels.hh` contains the header file for the CUDA kernels, Tutte, forcefield, dualisation, spherical projection.
  - `cu_array.hh` contains the header file for the `CuArray` structure.
  - `launch_ctx.hh` contains the header file for the `LaunchCtx` structure.
  - `isomer_batch.hh` contains the header file for the `IsomerBatch` structure.
  - `cuda_io.hh` contains the header file for the `IsomerBatch` I/O functions.
  - `isomer_queue.hh` contains the header file for the `IsomerQueue` structure.
- `benchmarks` Contains various scripts for benchmarking the forcefield, dualisation, Tutte and spherical projection implementations, as well as the parallel primitives, and timing of pipelines 0 through 7.
- `tests` Contains various scripts for testing and validating the forcefield, dualisation, Tutte and spherical projection implementations, parallel primitives, `IsomerQueue`, `IsomerBatch` and `CuArray` structures.

Chapter 2

Graphs and Forcefield Methods

2.1 Topology of Fullerenes

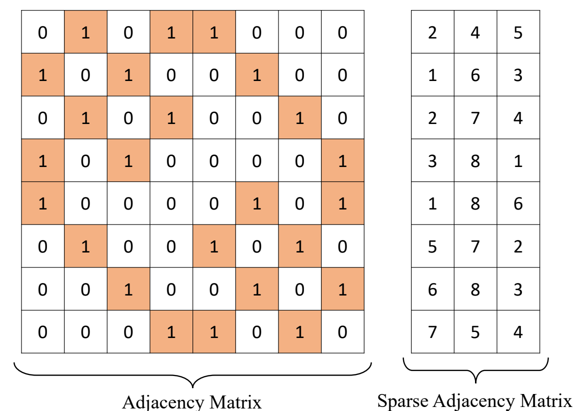
We have briefly summarised the developments in the *fullerene* program and forcefield calculations in general. The motivation for the speedup of said forcefield calculations and potential parallelisation paradigms that we might pursue. We must, however, cover a few more details about the underlying data structures that we will be working both in terms of computation and in terms of mathematical description.

2.1.1 Graphs

In graph theory a *graph* is a pair $G = (\mathcal{V}, \mathcal{E})$ where $\mathcal{V}$ is the set of *vertices* and $\mathcal{E} = \{\{u_1, v_1\}, \{u_2, v_2\}, \ldots, \{u_m, v_m\}\}$ is a set of all connected vertex pairs called *edges*.

If the set of edges $\mathcal{E}$ is comprised of ordered pairs $(u, v) \in \mathcal{V}^2$, we say the graph is a *directed graph*. We call u the source and v the target node, this ordering of pairs will become important later as we define canonical orderings and mappings.

Additionally, if a graph can be embedded in a plane, that is assign coordinates $\{x, y\} \in \mathbb{R}^2$ to all vertices $u \in \mathcal{V}$, such that all of its edges only intersect at their endpoints, the graph is said to be *planar*. There are infinitely many ways to embed a graph in the plane, but when the graph is planar and three-connected, any planar embedding will contain the same faces. Three-connected means: that at least three vertices must be removed from the graph to disconnect it.



0	1	0	1	1	0	0	0
1	0	1	0	0	1	0	0
0	1	0	1	0	0	1	0
1	0	1	0	0	0	0	1
1	0	0	0	0	1	0	1
0	1	0	0	1	0	1	0
0	0	1	0	0	1	0	1
0	0	0	1	1	0	1	0

Adjacency Matrix

2	4	5
1	6	3
2	7	4
3	8	1
1	8	6
5	7	2
6	8	3
7	5	4

Sparse Adjacency Matrix

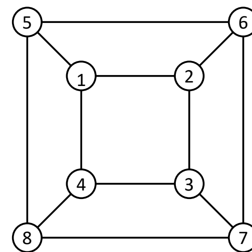


Figure 2.1: *An example showing a planar three-connected cubic graph and its corresponding adjacency matrix as well as a sparse representation.*

For these reasons a three-connected planar graph also has a well-defined set of faces $\mathcal{F}$ which allows us to represent the graph both in terms of edges and vertices but also as a set of faces, $G = (\mathcal{V}, \mathcal{E}, \mathcal{F})$. Ernst Steinitz proved that any convex polyhedron forms a three-connected planar graph, and every three-connected planar graph can be represented as the graph of a convex polyhedron, therefore three-connected planar graphs and by extension fullerenes are also known as *polyhedral graphs*.

We shall see later how we can construct 3D representations of fullerenes from their planar embeddings using a combination of *Tutte embeddings* and *Spherical Projection*.

In the context of Fullerenes we represent the carbon atoms as vertices and the chemical bonds between atoms as edges in the graph. A common way to store and represent the edges $\mathcal{E}$ and vertices $\mathcal{V}$ is through the *adjacency matrix* A_{ij}. The adjacency matrix is a square $n \times n$ matrix such that its element A_{ij} is one if there is an edge from vertex u_i to vertex u_j and zero otherwise. Obviously however since this matrix is incredibly sparse for a cubic (each vertex has 3 neighbours) graph, $sparsity = \frac{3N}{N^2} = \frac{3}{N}$ more precisely. We ought to store the connectivity in a sparse matrix format, the cubic nature of the graph lends itself to the $3 \times N$ *adjacency list*, here each list within the adjacency list describes the set of neighbours of a particular vertex.

Fig. 2.1 shows one such example of a cubic planar graph and its corresponding adjacency matrix and adjacency list representations. The visualisation makes the sparsity quite clear.

2.1.2 Generation of Fullerene Graphs

We wish to perform calculations on all possible fullerenes within a given isomerspace (C_N), therefore an exhaustive and efficient generator for all fullerene isomers of a given vertex number N is needed. Hasheminezhad et al.[7] defined a set of patch replacements or growth operations that could generate all fullerene isomers systematically, through structural induction, starting from either C20 or C28. These growth operations included a single operation that elongated a nanotube with minimal caps, and two classes of generalized Stone-Wales transformations that depend on one or two parameters. Brinkmann et al.[5] used these growth operations to define an efficient algorithm for generating all fullerene isomers up to a given maximum number. This algorithm prunes the recursion tree in such a way that only one representative of each isomorphism class is considered, preventing combinatorial explosion. The result is an incredibly efficient algorithm that has been used to generate an exhaustive database of fullerene graphs up to C400, which is available at the House-of-Graphs website.[1] The *BuckyGen* program is fast enough that the Brinkmann et al. made the perhaps reasonable claim "The generation cost is now likely to be lower than that of any significant computation performed on the generated structures.",[5] it will become clear that this is not the case.

2.1.3 Dual Graphs

While it is not the focus of this these to delve into the details of how these algorithms work, it is important to understand the underlying data structures that are used to represent the fullerene graphs. The *BuckyGen* program, which Brinkmann et al.[5] aptly named it, produces fullerenes in their face representation. This is also referred to as the dual or the graph $\boldsymbol{G^*}$. In graph theory, the dual of a graph is a graph that has a vertex for each face (region bounded by edges) in the original graph, and an edge connecting two vertices whenever the corresponding faces in the original graph share an edge. Note that the dual for fullerenes is a *triangulation* with 12 vertices of degree 5 and the remaining vertices with degree 6. The dual of a graph is not unique in general, but it is for cubic planar graphs, the same way the planar embedding is. The dual

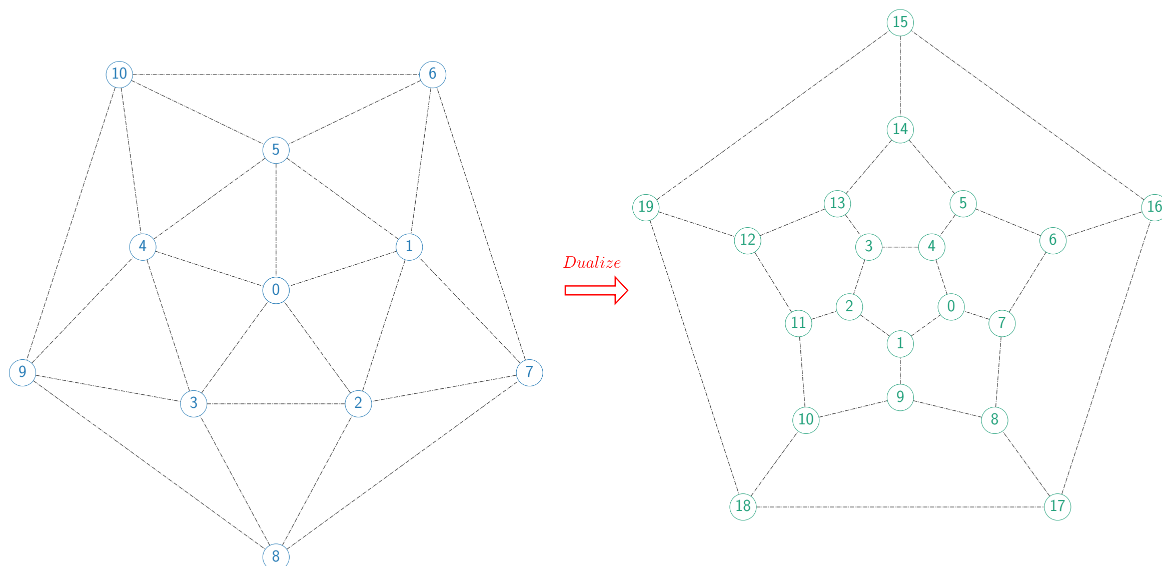


Figure 2.2: *Planar embeddings of (**left**) the dual representation (**right**) the cubic representation of the C_{20} dodecahedron. The numbering of neighbours shown in is chosen according to their canonical neighbour numbers generated by the* fullerene *program.*

operation is its own inverse hence $(G^*)^* = G$, this will become relevant later when we discuss the extension of forcefield methods to include flatness terms in the energy and gradient expressions. To intuitively understand the dual of a fullerene graph, consider the following example in Fig. 2.2, the C_{20} dodecahedron, which is the simplest fullerene graph.

From this figure we see how the 0th node in the dual corresponds to the face bounded by the edges $\{0, 1\}, \{1, 2\}, \{2, 3\}, \{3, 4\}, \{4, 0\}$ the edge $\{0, 1\}$ in the dual indicates neighbouring faces in the cubic and so on. The exact means of ascribing the canonical numbering of the neighbours will be further elaborated upon in the pipeline chapter where we discuss both the sequential *fullerene* program implementation and a corresponding parallel implementation (Section 5.3).

2.2 Forcefield Optimisation

Forcefield methods, currently provide one of the most computationally efficient methods of molecular-geoemtry-optimisation. Forcefield optimisation is a method which estimates the forces between atoms within molecules using parametrisation of e.g. bond-lengths, bond-bond angles, non-bonded interaction and out-of-plane torsion. The benefit of this functional form, is that it brings down computational complexity to either $\mathcal{O}(N^3)$, if non-bonded interaction is included, or $\mathcal{O}(N^2)$ otherwise. With N being the number of atoms in the molecule. Let us briefly summarise the developments of the forcefield method we have adopted in this thesis.

2.2.1 *Developments in Fullerene Forcefields*

The original version of the *Fullerene* program included a very simple harmonic forcefield of the form

$$E_{Wu} = \frac{k_p}{2}\sum_{i_p}^{p-edges}(R_{i_p}-R_p)^2+\frac{k_h}{2}\sum_{i_h}^{h-edges}(R_{i_h}-R_h)^2+\frac{f_p}{2}\sum_{j_p}^{60}(\alpha_{j_p}-\alpha_p)^2+\frac{f_h}{2}\sum_{j_h}^{3N-60}(\alpha_{j_h}-\alpha_h)^2 \tag{2.1}$$

Where k_p and k_h are the spring force constants for pentagons and hexagon edges respectively, R_p and R_h are the equilibrium bond lengths for pentagons and hexagons respectively, f_p and f_h are the force constants for pentagons and hexagons respectively, and finally α_p and α_h are the equilibrium bond angles for regular pentagons and hexagons respectively. This forcefield expression was introduced by Wu et al.[19] for $C_{60} - I_h$ specifically. And while the WU forcefield can in principle be applied to all fullerenes and usually yields structures which are in good agreement with optimized structures from quantum chemical calculations, it suffers from two problems. Firstly optimisations may converge to local minima if the initial structure is not close to the global minimum. Secondly the forcefield omits all torsion / dihedral bending terms and thus cannot reproduce the convexity or planarity of DFT optimized structures. These problems were addressed by Wirz et al.[18] who introduced a new and improved forcefield for fullerene optimisation. The Wirz forcefield is a generalization of the Wu forcefield and includes torsion terms. The Wirz forcefield is given by:

$$\begin{aligned}
E_{Wirz} = {} & \frac{f_{pp}}{2}\sum_{i_{pp}}^{pp-e}(R_{i_{pp}}-R_{pp})^2+\frac{f_{ph}}{2}\sum_{i_{ph}}^{ph-e}(R_{i_{ph}}-R_{ph})^2+\frac{f_{hh}}{2}\sum_{i_{hh}}^{hh-e}(R_{i_{hh}}-R_{hh})^2\\
& +\frac{f_p}{2}\sum_{j_p}^{60}(\alpha_{j_p}-\alpha_p)^2+\frac{f_h}{2}\sum_{j_h}^{3N-60}(\alpha_{j_h}-\alpha_h)^2+\frac{f_{ppp}}{2}\sum_{k_{ppp}}^{ppp-e}(\theta_{k_{ppp}}-\theta_{ppp})^2\\
& +\frac{f_{hpp}}{2}\sum_{k_{hpp}}^{hpp-e}(\theta_{k_{hpp}}-\theta_{hpp})^2+\frac{f_{hhp}}{2}\sum_{k_{hhp}}^{hhp-e}(\theta_{k_{hhp}}-\theta_{hhp})^2+\frac{f_{hhh}}{2}\sum_{k_{hhh}}^{hhh-e}(\theta_{k_{hhh}}-\theta_{hhh})^2
\end{aligned}$$

The energy expression of the Wirz and Pedersen forcefields are identical however Pedersen included a number of corrections to the gradient expression, which anecdotally has served to improve convergence towards global minima.

This forcefield includes torsion terms for all 4 possible face configurations (pentagon-pentagon-pentagon, pentagon-pentagon-hexagon, pentagon-hexagon-hexagon and hexagon-hexagon-hexagon) and also distinguishes between pentagon-pentagon, hexagon-pentagon and hexagon-hexagon bonds. Wirz showed that this forcefield produces structures with bond lengths, bond angles and torsion angles are in good agreement with DFT optimized structures for select fullerenes. The forcefield code is however written in difficult to read and maintain Fortran and is therefore not very extensible. This was the primary purpose and outcome of Pedersen's work:[12] To convert the Fortran forcefield code into readable and maintainable Python code; and to reformulate the forcefield optimisation problem in a massively data-parallel manner using vector operations computable in lockstep across not just singular isomers but potentially entire isomerspaces.

2.2.2 Bond Stretching

The bond stretching energy E_{bonds} is the energy required to stretch or compress a bond between two atoms by a distance $R_{ab} - R_0$ away from its equilibrium length R_0, where **ab** denotes the bond between atoms a and b. *Hooke's Law* states that the force required to stretch a spring by a distance x is proportional to x and is given by $F = -kx$ where k is the spring constant. The energy required to stretch a spring by a distance x is then given by $E = \frac{1}{2}kx^2$. The bond stretching energy is therefore given by Eq. (2.2) where k_R is the bond stretching constant.

$$E_{bond} = \frac{k_R}{2}(R_{ab} - R_0)^2 \tag{2.2}$$

Fig. 2.3 shows two atoms and their interatomic distance. The Wu forcefield[19] uses 2 different equilibrium bond length parameters one to signify that an edge is part of a hexagon R_h and one for pentagons R_p. The Wirz forcefield[18] accounts for the fact that every bond is a part of two and therefore decomposes the equilibrium bond length parameters into the three possible configurations: hexagon-hexagon R_{hh}, pentagon-pentagon R_{pp}, hexagon-pentagon R_{hp} each with their own corresponding force constants k_{hh}, k_{pp} and k_{hp}. It is not the scope of this thesis to derive the gradients of the energy expressions, we nonetheless provide the results in their vector calculus form as presented by Pedersen.[12] Eq. (2.3) shows the gradient of the bond

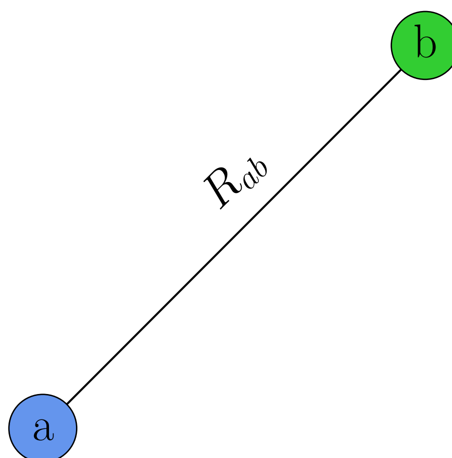


Figure 2.3: *Visualisation of bond stretching of two atoms.*

stretching energy with respect to the movement of atom a.

$$\nabla_a E_{bond} = -k_R(\|\boldsymbol{ab}\| - R_0)\widehat{\boldsymbol{ab}} \tag{2.3}$$

2.2.3 Angle Bending

Angle bending is the bending of the angle made up by three atoms a, b, c from their bonds, ab and ac. The energy is computed in much the same way that bond stretching is. We use the harmonic approximation of the energy expression around the cosine of the equilibrium angle $\cos(\theta_0)$. The energy constituent resulting from angle bending is expressed as follows:

$$E_{angle} = \frac{k_\theta}{2}(\cos(\theta_{abc}) - \cos(\theta_0))^2 \tag{2.4}$$

Fig. 2.4 shows the angle bending of three atoms. The Wu forcefield[19] uses 2 different equilibrium angle parameters one to signify that the atoms a, b, c are part of a hexagon θ_h and one for pentagons θ_p. The Wirz forcefield[18] and the Pedersen forcefield[12] both use cosines of the angle and equilibrium angle parameters with the cosine of the angle and equilibrium angle parameters respectively. This does not fundamentally alter the energy expression, but it avoids having to compute the inverse cosine which is both a somewhat computationally expensive operation but also numerically sensitive to floating point errors as it is only valid in the range $[-1, 1]$, so additional checks would have to be performed. The cosine of the angle is given by the dot product of the unit vectors of the bonds ab and ac and as such is both fast and numerically stable.

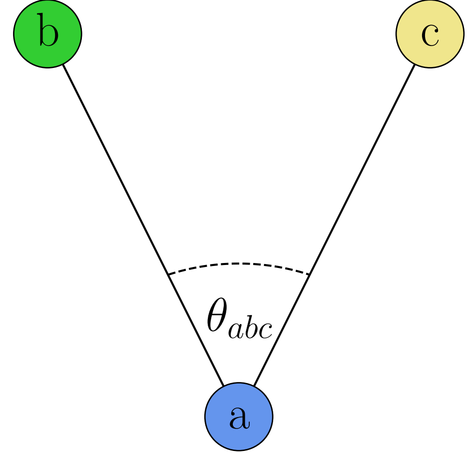


Figure 2.4: *Visualisation of angle bending of three atoms.*

The gradient of the angle bending energy, using cosines of angles with respect to the movement of atom a is given by Eq. (2.5).[12]

$$\nabla_a E_{angle} = -k_\theta(cos(\theta_{abc}) - cos(\theta_0))\left(\frac{1}{\|\boldsymbol{ab}\|}(\widehat{\boldsymbol{ab}}cos(\theta_{abc}) - \widehat{\boldsymbol{ac}}) + \frac{1}{\|\boldsymbol{ac}\|}(\widehat{\boldsymbol{ac}}cos(\theta_{abc}) - \widehat{\boldsymbol{ab}})\right) \tag{2.5}$$

2.2.4 Torsion / Out of Plane Bending / Dihedral Angle

The dihedral energy E_{dih} represents the energy required to move an atom a away from its optimal distance to the plane made up of the atoms $\boldsymbol{b}-\boldsymbol{c}-\boldsymbol{d}$. The dihedral energy is given by Eq. (2.6) where k_ϕ is the dihedral force constant, ϕ is the dihedral angle and ϕ_0 denotes the equilibrium dihedral angle.

$$E_{dih} = \frac{k_\phi}{2}(cos(\phi) - cos(\phi_0))^2 \tag{2.6}$$

Moving atom a changes the angle between the planes $\boldsymbol{a}-\boldsymbol{b}-\boldsymbol{c}$ represented by the normal vector $\widehat{\boldsymbol{n}}_{abc}$ and the plane $\widehat{\boldsymbol{n}}_{bcd}$. Naturally there exists a dihedral angle between the planes $\widehat{\boldsymbol{n}}_{acd}$ and $\widehat{\boldsymbol{n}}_{bcd}$ as well as between $\widehat{\boldsymbol{n}}_{abd}$ and $\widehat{\boldsymbol{n}}_{bcd}$, but symmetry dictates the expressions be the same. The gradient of the dihedral energy with respect to the movement of atom a is a rather convoluted expression and the reader is referred to the work by Pedersen,[12] for a full discussion of these gradient terms and their derivation.

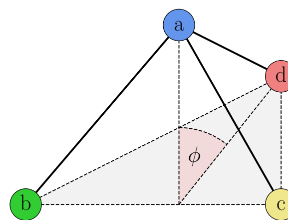


Figure 2.5: *Visualisation of the dihedral angle bending of the $\widehat{\boldsymbol{n}}_{abc}$ plane away from the $\widehat{\boldsymbol{n}}_{bcd}$ plane.*

2.2.5 Outer Gradient Components

So far we have considered components of the gradient expression related directly to the four atom system $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}, \boldsymbol{d}$, in Fig. 2.6 these atoms exist within the **inner circle** and the neighbours of the atoms $\boldsymbol{b}, \boldsymbol{c}, \boldsymbol{d}$ which are considered outer neighbours of atom **a** , reside within the **outer circle**. The inner angles are the angles $\theta_{abc}, \theta_{acd}$ and θ_{adb}. The outer angle contribution comes about from the fact that moving atom **a** also changes the angles $\theta_{bab_p}, \theta_{bab_m}$ as well as $\theta_{cac_p}, \theta_{cac_m}$ and $\theta_{dad_p}, \theta_{dad_m}$. The outer angle contribution is given by Eq. (2.7).

$$\boldsymbol{\nabla}_a cos(\theta_{bab_p}) = \frac{1}{\|\boldsymbol{ba}\|}(\widehat{\boldsymbol{bb}}_p - \widehat{\boldsymbol{ba}} cos(\theta_{bab_p})) \tag{2.7}$$

This expression is identical for all outer angles substituting atom **b** for **c** and atom $\boldsymbol{b}_p$ for $\boldsymbol{c}_p$ and so on. All the outer angles are outlined and labelled in Fig. 2.6. Both the Wirz forcefield and the forcefield by Pedersen included these outer angle gradient terms. Pedersen noted that not only outer angles are affected by the movement of atom **a** , the planes $(\boldsymbol{a}-\boldsymbol{b_p}-\boldsymbol{b_m})$, $(\boldsymbol{a}-\boldsymbol{c_p}-\boldsymbol{c_m})$ and $(\boldsymbol{a}-\boldsymbol{d_p}-\boldsymbol{d_m})$ too are affected. All dihedral angles involving these planes are affected and give rise to three additional terms in the gradient expression for each of these outer planes. These expressions are rather large so have been omitted here. The reader is referred to the work by Pedersen[12] for a full discussion of these gradient terms and their derivation. In algorithms, we shall

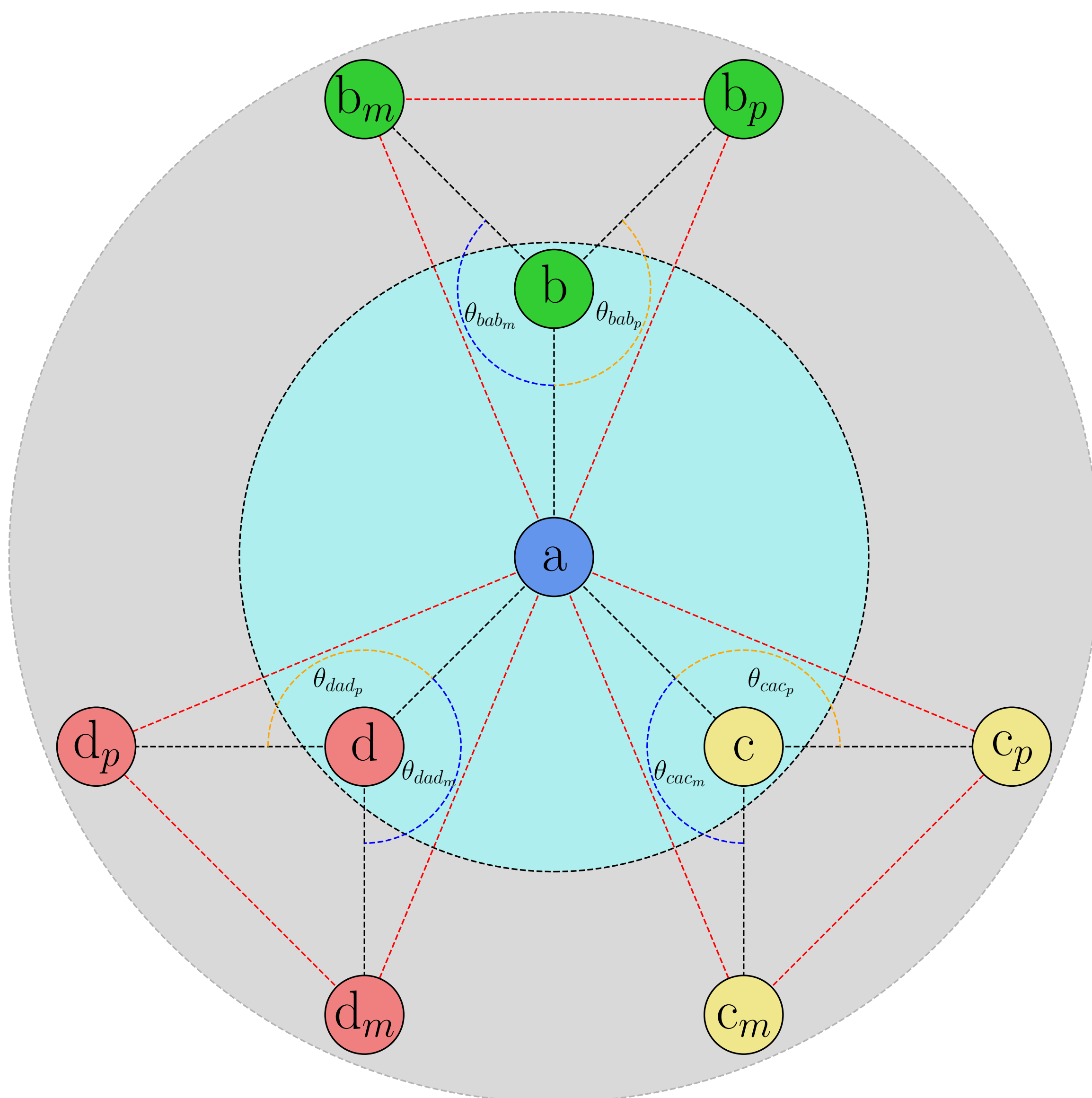


Figure 2.6: *The four atom systems* $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}, \boldsymbol{d}$ *resides within the* ***inner circle*** *and the neighbours of the atoms* $\boldsymbol{b}, \boldsymbol{c}, \boldsymbol{d}$ *are labelled* $\boldsymbol{b_p}, \boldsymbol{b_m}, \boldsymbol{c_p}, \boldsymbol{c_m}, \boldsymbol{d_p}, \boldsymbol{d_m}$ *these are considered the outer neighbours of atom a and reside within the* ***outer circle****. We developed this figure to help visualise all the components which are affected by the perturbation of atom* ***a*** *.*

refer to by them in their left-hand-side form $\nabla_a(\widehat{\boldsymbol{n}}_{bab_m} \cdot \widehat{\boldsymbol{n}}_{ab_mb_p})$, $\nabla_a(\widehat{\boldsymbol{n}}_{bb_mb_p} \cdot \widehat{\boldsymbol{n}}_{ab_mb_p})$ and $\nabla_a(\widehat{\boldsymbol{n}}_{bab_p} \cdot \widehat{\boldsymbol{n}}_{ab_mb_p})$ for brevity.

2.3 Summary

In Section 2.1 we have given a brief exposition on Graphs and how they relate *Fullerenes* with nodes representing atoms and edges representing bonds, we have shown how adjacency lists are an efficient representation for cubic graphs in particular. We mentioned how there are two information equivalent ways to represent a fullerene, either as a cubic or as a dual this will become relevant in Section 5.3 and Section 4.7.

In Section 2.2 we have given a summary and explanation of the energy terms (bond-stretching, angle-bending and dihedral bending) and corresponding forces involved in the harmonic forcefield approximation used to optimise fullerene geometries.

Chapter 3

Hardware

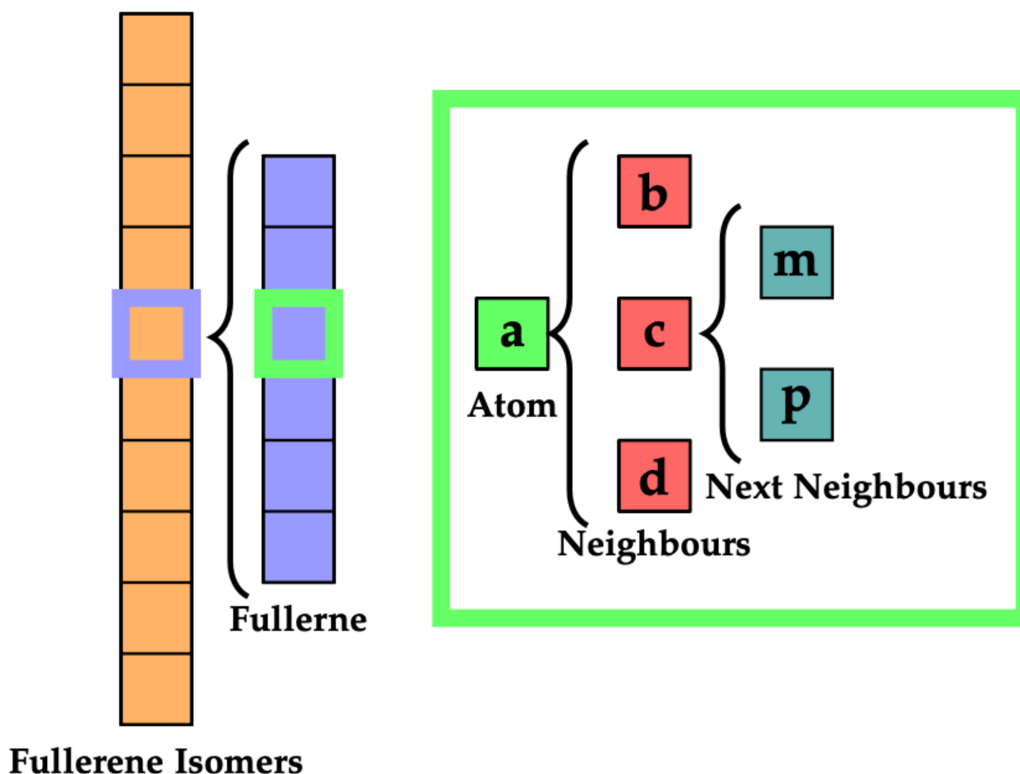


Figure 3.1: *Visualisation of the isomerspace optimisation problem. From left to right: Each square in the (**orange**) column represents a specific fullerene in the isomerspace. The (**blue**) column represents the optimisation of a single fullerene. Each square (**green**) in the fullerene represents a node / carbon-atom. Three neighbours (**red**) are assigned to each carbon atom and two outer neighbours (**teal**) to each of the neighbours.*[12]

Pedersen prepared the forcefield optimisation problem for parallelisation through wide vector (`Numpy`) operations. He explicitly showed how isomerspace optimisation can be decomposed into multiple levels of parallelism (Fig. 3.1). It is our goal to investigate to what extent it is possible to exploit this parallelism on modern hardware.

There are a number of ways one could exploit this parallelism. A common way to parallelise is to delegate independent tasks to individual processing elements. While task parallelism is the right solution to many problems and can be implemented efficiently, there are other parallelisation paradigms more suitable for vector-based operations. One such paradigm that *is* particularly suited for vector problems is the *Lockstep Parallel* paradigm. Lockstep Parallelism refers to the process of applying the same operations (same instructions) to every constituent of a vector simultaneously. One benefit of lockstep parallelism is that synchronisation is abstracted away from the programmer, since a lockstep-parallel vector engine is conceptually synchronous, in theory this affords close to perfect utilisation of compute units. Naturally, some hardware architectures approximate this ideal better than others.

For general purpose programming CPUs are the most common choice. CPUs are good at exploiting task-level parallelism since they follow the *multiple instruction multiple data* (MIMD) model, the CPU is optimised for performing entirely independent work on each of its cores. CPUs only spend approximately 5-10%[8] of transistors on actual computation (ALUs) the rest is spent on e.g. caches, branch prediction, cache control and out of order execution. Modern accelerator chip architectures instead have far more compute units, but without the sophisticated control-flow management. E.g. TPU & NPU (*Tensor Processing Unit*, *Neural Processing Unit*), which are optimised for tensor contractions and matrix multiplications, GPGPUs (*General Purpose Graphics Processing Unit*) optimised for graphics and vector operations, CGRA (*Coarse Grained Reconfigurable Architecture*) which are a customisable grid of tightly interconnected processing and memory elements, AMD Xilinx AI Engine, a grid of tightly connected vector processing units.

Each of these architectures are massively parallel and offer extremely high *theoretical* peak performance, but this is rarely achieved. To reach this performance the problem and the implementation must have the specific shape that fits the hardware, as the many simple compute cores do not operate independently like on a CPU. However, common to all these architectures is that a broad vector engine is a fitting abstraction. Therefore, if we can write an algorithm in *lockstep parallel*, we reach near peak performance, given data locality.

Currently, CPUs have in the range of 4-64 cores whereas modern vector engines have thousands. Thus while CPUs are good for general purpose compute and are easy to program, there is a huge potential to be exploited. Of these accelerator architectures GPGPUs are by far the most mature and widely accessible. Thus, this thesis will focus on implementation of lockstep parallel algorithms on GPGPUs. We will pick apart the forcefield-optimisation and describe how this computation can be mapped to any data parallel hardware and how we can move from the abstract parallel lockstep algorithm to a concrete and massively efficient implementation. We need to delve deep into the details of hardware architecture and specific details of the programming model to achieve this, covering memory hierarchies, register pressure, access patterns, bank conflicts, occupancy, parallel primitives, asynchronous execution and more.

3.1 GPGPUs

The most common GPGPU execution model is the *single instruction multiple data* (SIMD) model. SIMD is a form of data parallelism where a single instruction is executed on multiple data elements simultaneously, using wide vector registers and *arithmetic logic units* ALUs. SIMD is a common execution model particularly on GPUs but also exists in small-scale (128-512 bit wide) on modern CPUs. Some GPGPUs instead use *single instruction multiple threads* (SIMT) which is a form of data parallelism where a single instruction is executed on multiple threads simultaneously. SIMT is the execution model used by Nvidia GPUs and resembles the SIMD model with the exception that each thread

has its own registers, stack pointer and program counter. This makes it a more flexible execution model than SIMD but ultimately the two are very similar.

Not all workloads are suited for GPUs however: Codes with a lot of branching, many conditional statements or random memory access patterns will not perform well. Furthermore, it is in general rare for GPU applications to meet these theoretical peak performance throughput figures, many HPC workloads simply require too much access to memory and so it is not uncommon for GPU workloads to achieve on the order of 5% of their theoretical peak performance. It all hinges on our ability to write the forcefield calculation in such a way that all or almost all calculations are executed in lockstep across entire isomerspaces and to ensure that data locality is maximised, this is the premise for good utilisation of the GPGPU hardware.

3.1.1 Choosing a Programming Model

Graphics processing units have been around for a while however they have traditionally not been used for anything other than graphics rendering. The advent of General-Purpose computing on Graphical Processing Units (GPGPU) frameworks, specifically Compute Unified Device Architecture (CUDA) and OpenCL gave rise to scientific computing on GPUs. Today a plethora of GPGPU frameworks exist for porting code to GPUs, from compiler directive standards like `OpenACC` and `OpenMP` to just-in-time (JIT) compilers like `Numba`.

Given that the implementation by Pedersen was implemented in Python/`Numpy`, a natural choice would be to parallelise using either `Numba`, `CuPy` or any of the numerous other automatic parallelisation frameworks. Higher-level approaches lack exposure to lower level language features but most importantly they do not give the user control over data locality and memory management, which is *crucial*. Additionally, composing a program out of smaller premade kernels, while flexible, would not permit whole program optimisation or link-time-optimisation (LTO) otherwise afforded by writing and compiling the entire source code. For these reasons, this project was developed entirely using CUDA and C++.

3.1.2 CUDA programming model

Let us introduce some concepts necessary to discuss parallelism on CUDA-enabled GPUs. The CUDA programming model provides an abstraction of GPU architecture that acts as a bridge between an application and hardware implementations. CUDA is available in 3 flavours C/C++, Fortran and recently Python. The Python version is currently a subset of the full C++ / Fortran implementation and the *kernels* are written as strings in a C-like syntax.

We will often refer to the GPU as the *device* available on the system and the CPU is referred to as the *host*. To run a CUDA program we must first: transfer data from host memory to device memory since most Nvidia GPUs are dedicated processors with their own physically separate memory. Second: load the GPU program and execute. Lastly:

copy the results from device memory to host memory.

The nature of a CUDA kernel, a function that is launched from the host and executed on the device, is such that anything you write is executed by all $N_{threads}$ different threads as opposed to only once by a single thread, like regular C/C++ functions. Listing 3.1 and Listing 3.2 show how an array wide AXPY (*A times X Plus Y*) might be implemented in C++ and CUDA respectively. Notice how the for loop is replaced by a single line of code that is executed by all threads. The `blockIdx.x` and `threadIdx.x` are that are built-in thread-local (each thread has its own value of these) variables.

```
void saxpy(...){
for(int i = 0; i ¡ N; i++){
    c[i] = a * x[i] + y[i];
}
}
```

Listing 3.1: *Standard C++ sequential AXPY function*

```
__global__ void saxpy(...){
int tid =
threadIdx.x + blockIdx.x *
blockDim.x;
c[tid] = a *
x[tid] + y[tid];
}
```

Listing 3.2: *Typical thread parallel AXPY implementation*

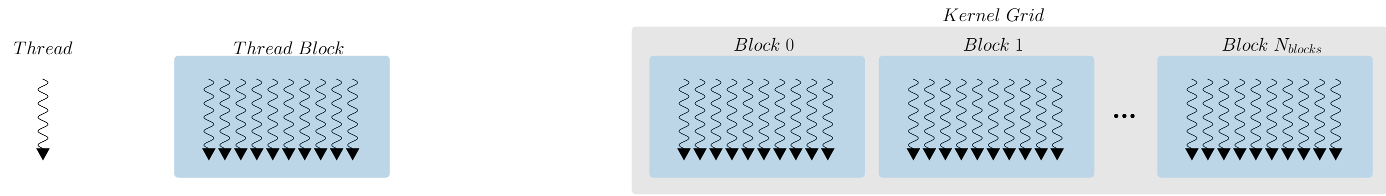
While we can sometimes make do with the notion of a global thread identifier the fact is, the Nvidia GPU has multiple levels of both memory and parallelism inherent to the hardware. This tiered approach is not exclusive to Nvidia GPUs, AMD also uses multiple tiers and many CPUs implement some form of Non-Uniform Memory Access (NUMA), which requires special attention to maximise performance.

Specifically, the CUDA programming model currently exposes three levels of parallelism each mapping to a level of the hardware stack. See Fig. 3.2.

- Threads are the lowest level of parallelism their instructions are carried out on a single core, and they have access to fast thread-private memory *registers*.

- Blocks are collections of threads, the size of blocks is determined when you launch a kernel. Blocks reside on a single *Streaming Multiprocessor* (SM) which is a collection of primitive cores and control flow logic. Blocks have access to a form of low-level cache *L1 Cache* a section of this cache can be explicitly accessed by the programmer and is known as *Shared Memory*.

- Lastly, we have the kernel grid or just grid, a grid consists of blocks and can be thought of as running on the entire device. Grids have access to the slowest form device of memory in the device memory hierarchy, the GDDR SDRAM (*Graphics Double Data Rate Synchronous Dynamic Random Access Memory*) referred to as *Global Memory* since it can be viewed and modified by all threads on the device.



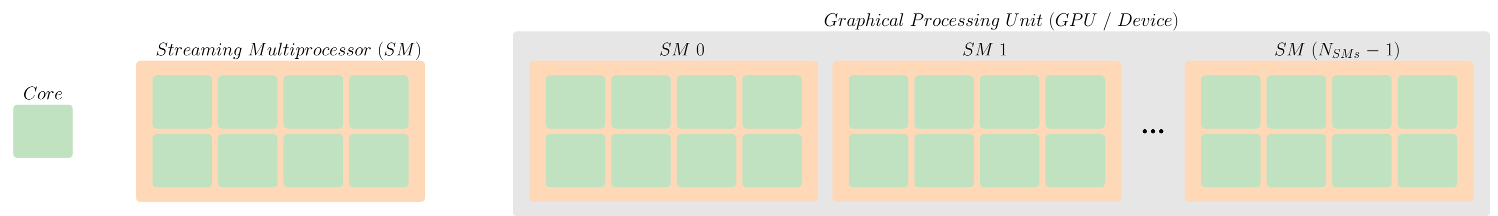


Figure 3.2: *A view of the three levels of parallelism in the CUDA programming paradigm. Each tier of the software stack, Thread, Block and Grid has an approximate mapping to* `Nvidia` *hardware and memory.*

There are some important caveats to the underlying structure of thread blocks, operations on Nvidia GPUs are performed on thread groups known as *warps*, consisting of 32 threads, AMD calls them *wavefronts*, which are made up of 64 threads. Even though it is technically possible to pick block sizes in the range $N \in \mathbb{N} \cap [1, 1024]$ picking any N that is not an integer multiple of 32 will result in masking away excess threads.

$$N_{ghost_threads/block} = 32 - (N_{threads/block} \mod 32) \tag{3.1}$$

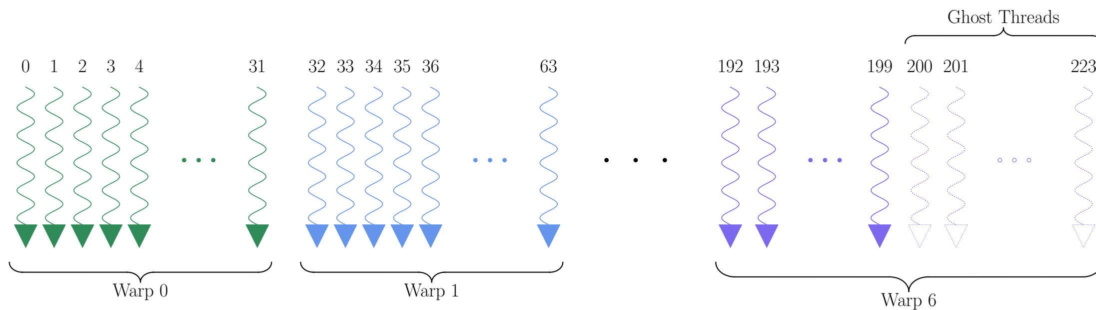


Figure 3.3: *Schematic of the underlying structure of a thread block of size 200. The block consists of 7 warps each containing 32 threads, the last 24 threads are masked away in CUDA.*

While CUDA exposes thread-level parallelism, and we may use thread identifiers to index arrays to perform parallel operations, one should take great care when using

these thread IDs for control flow. Due to the Single Instruction Multiple Thread (SIMT) nature of Nvidia GPUs, any program branches must be executed by all threads within a warp. Thus, if we create branches that require threads within the same warp to perform different operations, the instructions will be serialized and threads not participating in an instruction are simply masked away.

3.1.3 Discrete Resource Allocation

There are some important performance implications caused by the discrete allocation of resources on the GPU. Let us consider the primary 3 components that lead to odd saw-like tendencies in later benchmarks.

Block size: Each architecture iteration specifies a maximum number of threads that can reside on a single SM. This number has so far been either 1024, 1536 or 2048. One might reasonably assume that, given a thread-limit of 1024, one can fit 5 blocks of block-size 200. This is not the case, instead, threads are allocated in chunks known as the *Warp Allocation Granularity*, this in combination with the maximum number of warps per multiprocessor therefore only 4 blocks of size 200 can be allocated on each SM. More generally if we assume that register allocation (40 registers per thread) and shared memory allocation are not limiting factors, the effect on Occupancy as Nvidia coins it and Occupancy accounting for wasted effective compute due to ghost-threads, is depicted in Fig. 3.4. We also plot the occupancy and effective occupancy as with registers per thread set to 64, as is the compilation option of choice for much of the program. Notice how the effective occupancy has fewer yet sharper jumps in occupancy whereas the occupancy numbers vary at 32-thread intervals. These numbers are derived using the CUDA occupancy calculator.[15]

Registers per thread: Nvidia permits two ways of controlling the maximum number of registers allocated to each thread in a kernel launch. One either specifies a compilation flag that applies across the compilation unit, or attaches a launch bound attribute to kernels as needed. If neither is used the compiler has free rein to determine register allocation for each kernel. The register file size varies from architecture to architecture but has been fixed at 2^{16} registers per SM since the Maxwell architecture (2014). Using the same occupancy calculator as previously we can compute occupancy as a function register count per thread. We fix the block size at 128 and shared memory per block at 0 MB. Results are plotted in Fig. 3.7.

These results are very convenient as they give us good intuition for which register limits to test in future benchmarks. We note that there are specific breakpoints of maximum registers per thread, however Fig. 3.7 only demonstrates these effects for block size 128, therefore we need to investigate the effect of changing maximum register in conjunction with block size. As per previous discussion we need only consider block sizes that are integer multiples of 32 since they are composed of warps.

In Fig. 3.6 we illustrate the effects on occupancy that changing block size and registers per thread, carry. While occupancy may very well change on every block size increment

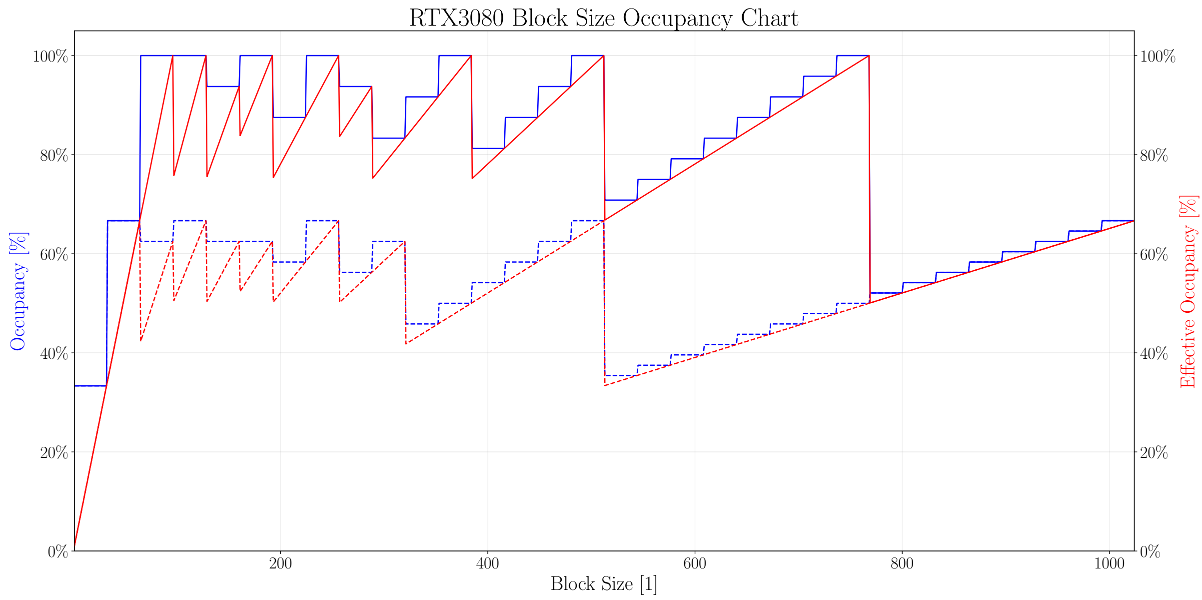


Figure 3.4: *Occupancy (**blue**) and Effective Occupancy (**red**) as a function of block size on an `RTX3080` GPU. Register count is set to 40 (**solid lines**) / 64 (**dashed lines**) registers per thread and shared memory is set to 0 MB per block, to remove these variables from the equation.*

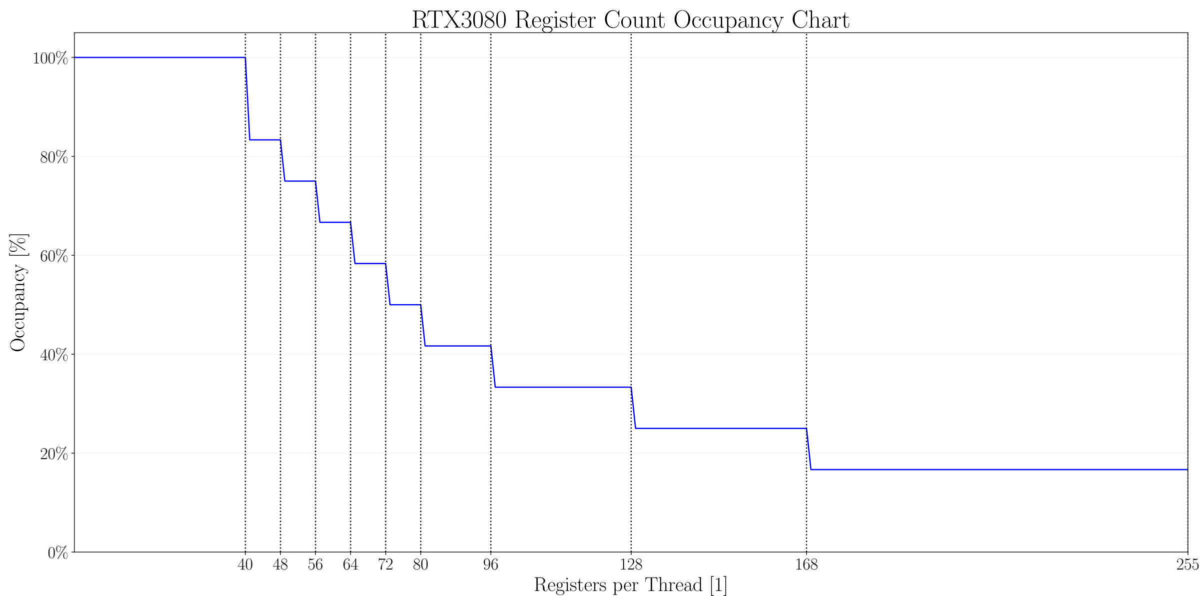


Figure 3.5: *Occupancy (**blue**) as a function of register count per thread on an `RTX3080` GPU. Block size was fixed at 128 and shared memory is set to 0 MB. The **black** lines indicate break points where maximum number of registers per thread are allocated for a given occupancy level.*

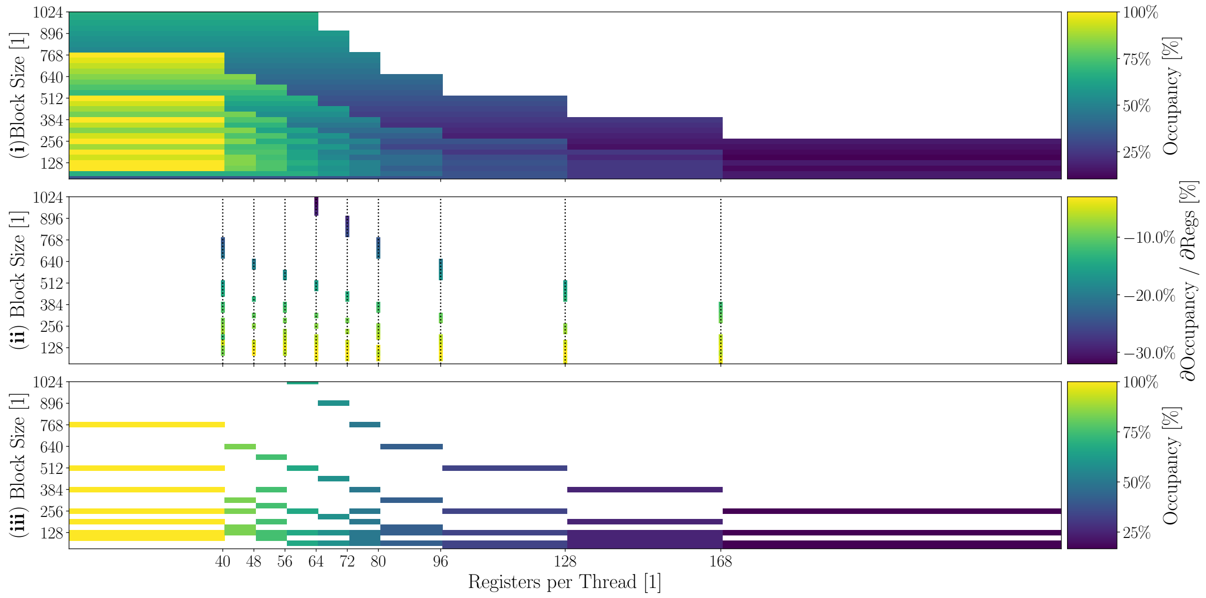


Figure 3.6: *(**i**) Colormap of Occupancy as a function of both maximum registers per thread and the block size. The (**ii**) subplot shows the forward difference derivative of the occupancy matrix (**i**) in the x-direction. The (**iii**) subplot shows which block sizes have the highest occupancy for a given register count.*

of 32 threads, registers per thread only affect occupancy at specific points. To make this perfectly clear we apply the forward difference method to find the non-zero derivatives of occupancy with respect to registers-per-thread (RPT). We find negative $\partial Occ/\partial RPT$ at RPT levels $40, 48, 56, 64, 72, 80, 96, 128, 168$ indicating that these are the thresholds of interest that ought to be tested to optimise performance of our kernels. Furthermore, part Fig. 3.6(**iii**) shows areas of maximum occupancy with respect to block size. What is notable here is that block-size 128 is the only block size which for any given RPT value has the best occupancy. We will use this information when deciding on a problem size to profile our forcefield kernels with, in Fig. 4.18.

Shared Memory: The amount of shared memory required per block can either be allocated statically declared by using the `__shared__ int array[64]` syntax or, as is often the case, dynamically by specifying it in the host code and accessing it using the `extern` keyword. The amount of available shared memory per SM has evolved a fair bit in recent architectures. Specifically the `RTX 3080` is equipped with 102400 bytes of shared memory per SM. It follows, then, that requesting a lot of shared memory can limit the amount of blocks that can launched. We can similarly plot compute the occupancy levels as a function of shared memory per block. Computed occupancy levels and breakpoints are shown in Fig. 3.7. Note that these numbers are highly block size dependent and in this example block size was fixed at 128 and registers per thread at 40

The actual occupancy in any real application will depend on all three variables, block-size, registers per thread and shared memory per block. However, sometimes, and the case in this thesis, the amount of shared memory required per thread is fixed as memory complexity is linear $\mathcal{O}(N)$. Block size and register count are not possible to

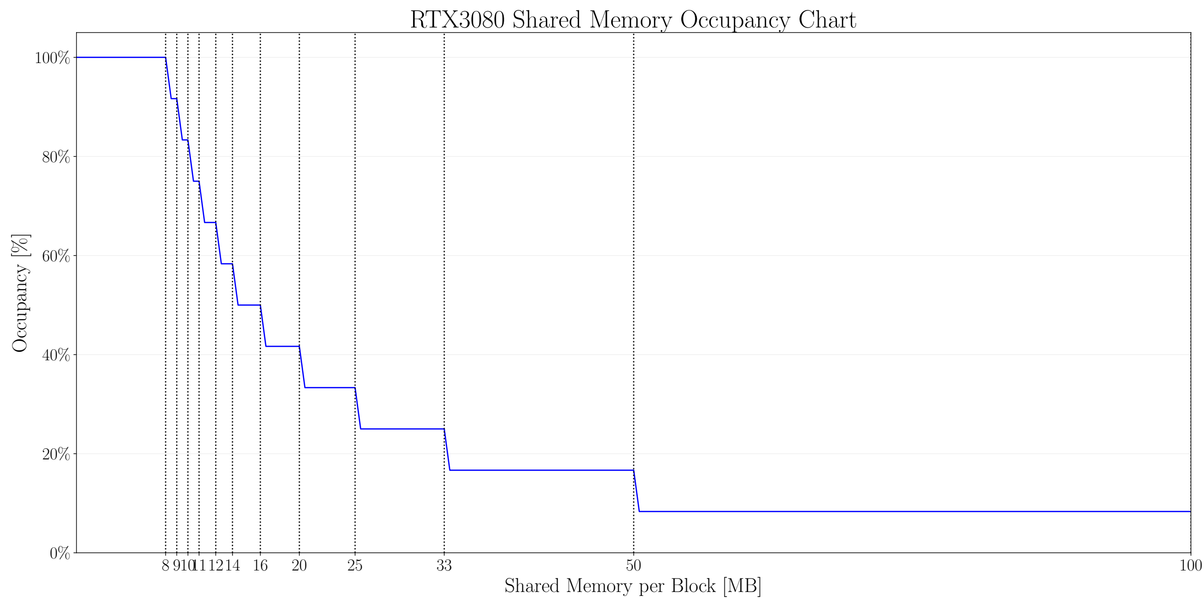


Figure 3.7: *Occupancy (**blue**) as a function of shared memory usage per block on an* `RTX3080` *GPU. Block size was fixed at 128 and register count per thread at 40. The (**black**) lines indicate break points where maximum shared memory is allocated for a given occupancy level.*

decouple however so there is some inherent trade-off in choosing a register per thread - count which favours particular block-sizes. We shall see this in detail when measuring performance of the forcefield calculations.

The performance implications of SIMT and the warp-level discretization of resource allocation will become apparent when benchmarking the GPU applications.

3.1.4 Throttling

We have just seen how choice of maximum register count, shared memory per block and block size all affect occupancy and effective occupancy, we have yet to discuss however, the impact of throttling.

GPUs and especially modern ones, do not run at fixed clock speeds, instead the driver determines when the GPU should boost the clock speed and conversely reduce it when the GPU is overheating. The advantage of course being that better cooling will allow us to squeeze out more performance from a given device. The disadvantage is that reliability of performance measurements are slightly compromised, or at least require extra care to be taken.

In order to determine the effects of variable clock speed or *throttling* we conduct an experiment where a forcefield computation is carried out on the GPU repeatedly for a total of 2000s. We then bin these kernel runtimes into 10s intervals and plot the distribution of kernel runtime as a function of elapsed time. Fig. 3.8 shows the result of this, it becomes clear that Device 0 is both more stable and more performant than Device 1, indicating better thermal conditions for Device 0. Furthermore, using both

devices at the same time appears to produce more erratic kernel runtimes.

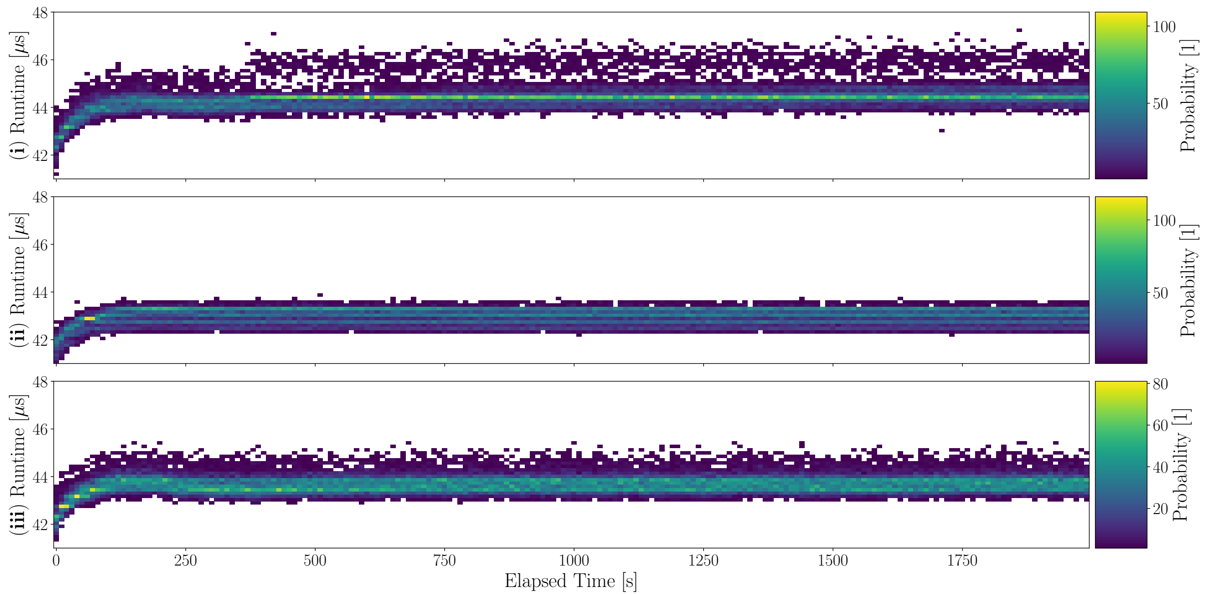


Figure 3.8: *Forcefield Optimisation kernel runtime as a function of elapsed time, the colour represents the probability of a certain runtime at a given elapsed time. Kernel runtimes are binned into 200 x-axis and 50 y-axis segments. These distributions are shown for* (**i**) *Device 0 and 1 running simultaneously,* (**ii**) *Device 0 and* (**iii**) *Device 1.*

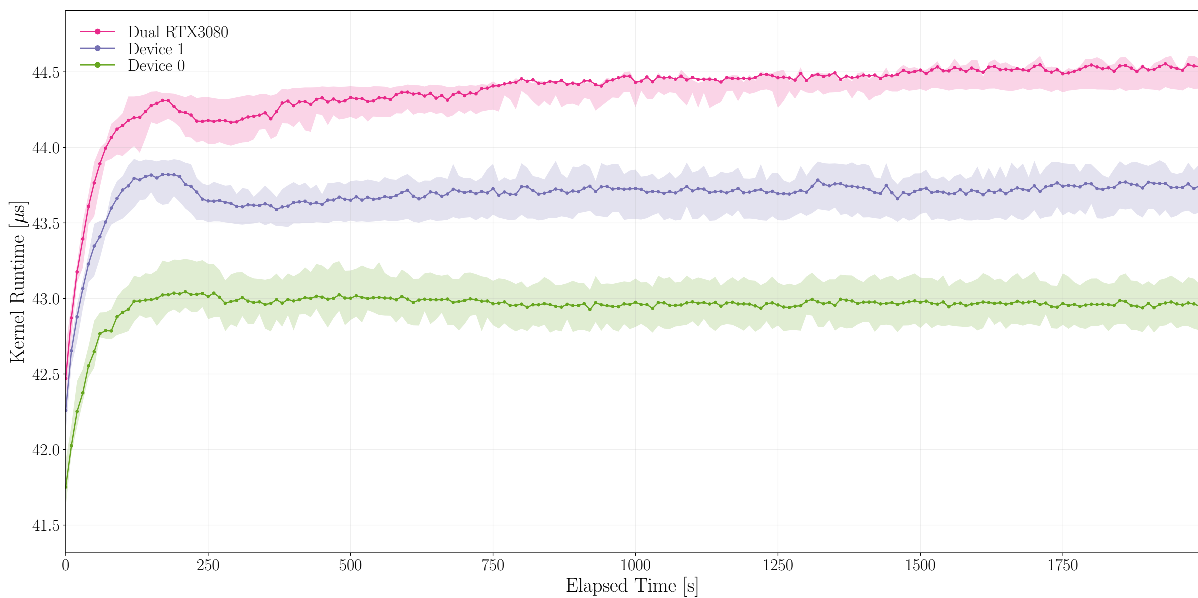


Figure 3.9: *Mean Forcefield Optimisation kernel runtime for* (***pink***) *Device 0 and 1 running simultaneously,* (***blue***) *Device 1 and* (***green***) *Device 0. The middle 68% of kernel runtimes are represented by the shaded areas.*

In Fig. 3.9 we take the mean kernel runtime and plot the middle 68% of kernel runtimes. From this we are able to clearly see the performance differences as well as the much longer stabilisation period of both Device 1 and the two devices together. It would

seem that the runtime distribution for dual GPU continues to change even after 2000s of elapsed time. Device 0 on the other hand appears to have stabilised already after approximately 200 seconds. We therefore choose to run all single kernel benchmarks on Device 0 and precede each of these benchmarks with 200s of GPU utilisation.

3.2 Memory

In the early days, computers were much simpler. The various system components CPU, memory, mass storage, networking et cetera were developed in tandem consequently performance was quite balanced. Importantly, networking and memory were not much faster than the CPU at delivering data.[6]

This balance was disrupted as the basic structure of computers stabilised, and hardware developers optimised individual subsystems. Performance of some components fell behind particularly mass storage and memory subsystems (SDRAM today). To corroborate this we look at the memory and compute performance of GPGPUs from the last 17 years (Fig. 3.10). As we might expect the development of memory bandwidth as well as compute throughput both exhibit exponential behaviour. The much famous *Moore's Law*, the empirical observation that transistor density doubles roughly once every two years, led to the prediction that computer performance doubles every 18 months.[11] From the exponential fits in Fig. 3.10, we derive doubling rates of $2.6Yr$ for compute and $5.3Yr$ for bandwidth, indicating that this performance doubling rate has reduced somewhat. Furthermore, we realize the stark contrast between improvements in memory bandwidth and compute throughput. Indeed, it would seem that the number of operations we must perform for every byte transferred is doubling every $\frac{ln(2)}{0.27-0.13} = 4.9Yr$.

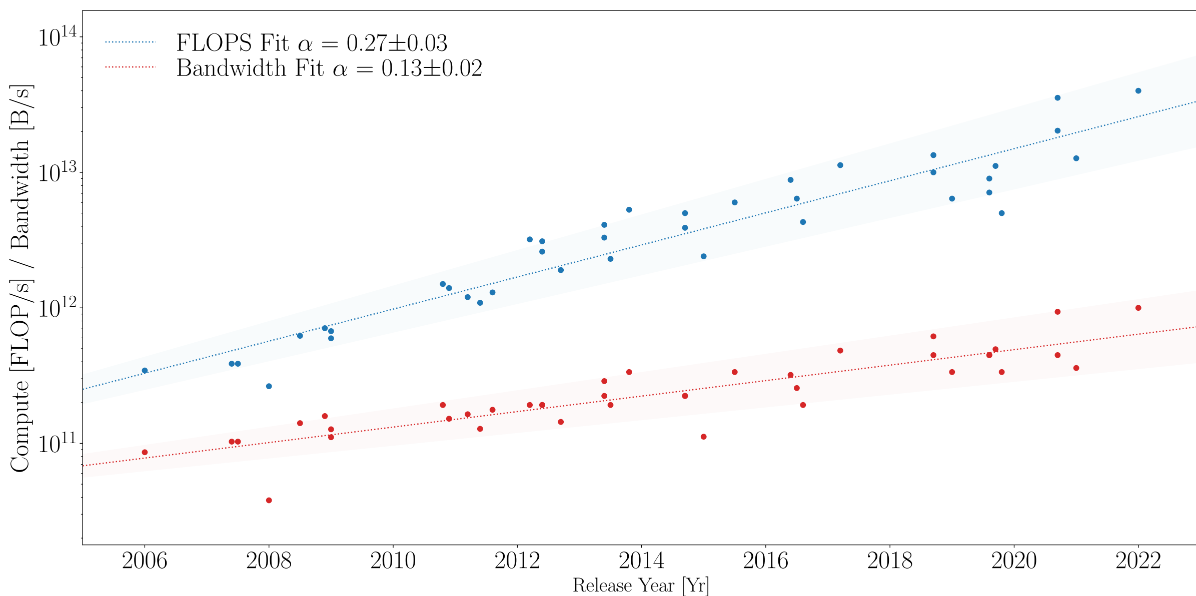


Figure 3.10: *A comparison of compute throughput in FLOP/s and DRAM bandwidth for consumer grade GPUs from 2006 to 2022. A linear fit to the natural logarithm of these metrics show exponential coefficients of* 0.27 *and* 0.13 *respectively. The uncertainty shown is* 1σ *derived from fitting parameter covariance matrix*

3.2.1 Memory Hierarchy

As mentioned the Nvidia GPUs have a three-level memory hierarchy, and as a general rule of thumb, the higher the level of the memory hierarchy the slower the access time. With GPUs it is common to get bandwidth bound if you need to frequently access DRAM, it is therefore vitally important that we consider the actual placement of our data in memory. We will see later that the penalty for fetching data from DRAM rather than registers or shared memory can be quite severe.

3.2.2 Arithmetic Intensity

Let us introduce the term *Arithmetic Intensity* I as the ratio between work W in FLOPs to memory traffic Q Bytes/s:

$$I = \frac{W}{Q} \tag{3.2}$$

We often use this number to gauge the theoretical limits of performance for a given application, this is done through roofline analysis. Roofline plots are rather odd-looking plots designed to show the performance of applications relative to what is theoretically possible.

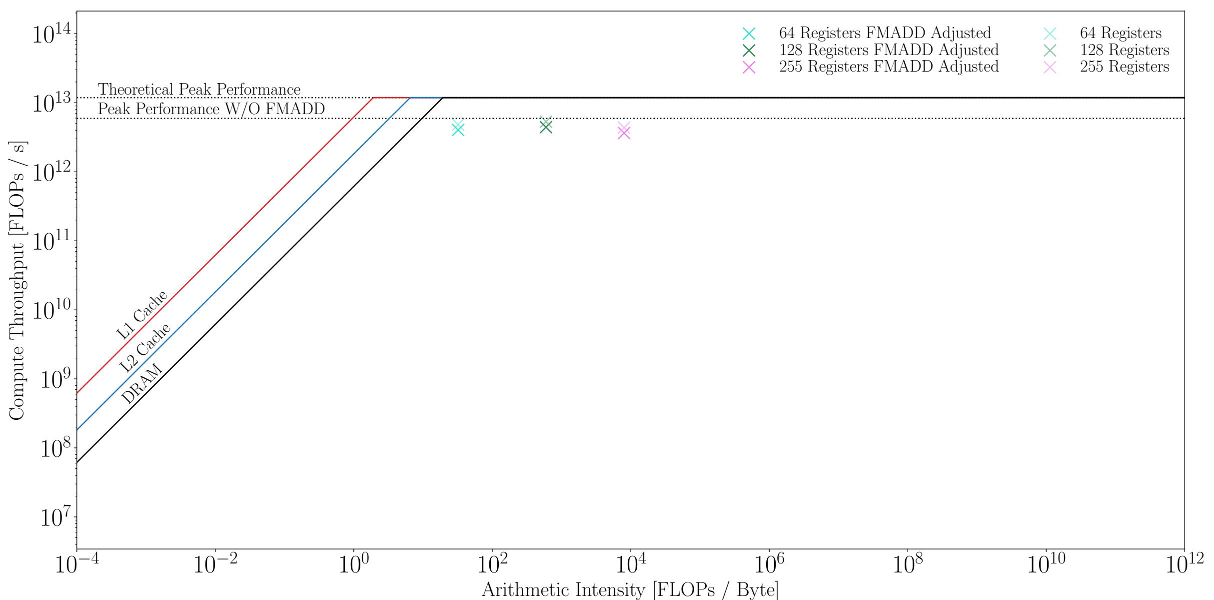


Figure 3.11: *Roofline analysis for the `RTX 3080` GPU cache bandwidths were deduced from profiling results as these numbers are not publicly available. The plot shows a roofline for level of the Nvidia memory hierarchy.*

Let β be the peak bandwidth, the roofline function is then defined as Eq. (3.3)

$$\text{Roofline} = min\{I \times \beta, \boldsymbol{\pi}\} \tag{3.3}$$

To achieve theoretical peak performance on many modern-day GPUs, we require all operations to be Fused-Multiply-Add operations, which technically count as 2 FLOPs.

This is rarely the case so for that reason we provide both this unrealistic ceiling that the vendor provides and a modified ceiling where we weigh FMADDs as a single operation. There is a myriad of other technical details that complicate roofline analysis. Most urgently, arithmetic intensity is determined from reads and writes to DRAM, however, it is possible to achieve higher effective bandwidth through caches. Therefore, it can be instructive to include rooflines for more levels of a memory hierarchy using, β_{DRAM}, β_{L1} and β_{L2}. It follows that for $I \times \beta \leq \boldsymbol{\pi}$ the program is limited by bandwidth, *memory-bound*, and if $I \times \beta \geq \pi$ the application is *compute-bound*.

3.2.3 Register Pressure

We will see later that, in spite of accounting for different memory hierarchies arithmetic intensity can be artificially influenced by register pressure. The reason for this is that, by reducing the number of registers available to each thread, the compiler is forced to spill more data to memory. If your kernel has low register pressure, the compiler may only spill data to L1 cache or L2 cache. If your kernel has high register pressure, the compiler may be forced to spill data to DRAM causing extreme performance degradation. (See Fig. 4.16 for an example of this).

3.3 Benchmarking Methodology

In this thesis we will benchmark numerous algorithms and variants of these as well as processing pipelines, therefore we provide here the measures taken in all benchmarks. As per the discussion in Section 3.1.4 prior to all benchmarks we heat up the GPUs by running arbitrary kernels for 200seconds, as this appeared to be sufficient given figure Fig. 3.9. To make sure the results are consistent we run all benchmarks 10 times and record the mean and standard deviations of the runtimes. Standard deviations are presented as shaded grey areas around the mean lines in the benchmark figures.

For certain algorithms, particularly those where an indeterminate number of iterations is required for convergence, it is important that the isomers are sampled randomly from the isomerspace. And perhaps more subtly while one might not think that this matters in lockstep algorithms with fixed number of operations and iterations, the actual graph layout of a given isomer changes the access pattern to shared memory inevitably causing varying levels of bank-conflicts.

By design the generator `BuckyGen`[5] found in the *fullerene* program produces isomers in procedural and recursive fashion. Thus sampling e.g. the first 1000 isomers will not produce a representative sample of the isomers present in that isomerspace. To get around this we have developed a random isomerspace sampling algorithm, shown in Algorithm 1.

The algorithm works by first producing a random list of N_s non-repeating integers in the range from 0 to M_{C_N}, once we have such a list $\boldsymbol{R}_I$ we sort it in ascending order such that accessing the jth gives the jth the smallest ID. Now we start iterating over the

isomerspace procedurally generating isomers and appending them to $\boldsymbol{I}_l$ if the next random number in the sequence $\boldsymbol{R}_I[j]$ matches the iteration number i. Random numbers were produced in the implementation using `std::shuffle()` and the Mersenne twister engine, random generator.[10]

Algorithm 1 Random Isomerspace Sampling Algorithm

```
M_{C_N}                                   ▷ Number of isomers in isomerspace C_N
N_s                                       ▷ Number of samples to pick from isomerspace
R_I ← random_uniform(0, M_{C_N})          ▷ Uniform distribution of integers from 0 to M_{C_N}
R_I^* ← R_I[0, ··· , N_s − 1]
R_I^* ← sort(R_I^*)                       ▷ Sort the list in ascending order
I_l                                       ▷ N_s × 1 List of randomly sampled isomers
j ← 0
for i ∈ {0, . . . M_{C_N} − 1} do
    G ← buckygen_next()
    if i = R_I^*[j] then
        I_l[j] ← G                        ▷ Add isomer to list
        j ← j + 1
    end if
end for
```

We produced a random sample of $\mathsf{min}(10000, M_{C_N})$ isomers from each isomerspace $C_{20}, \cdots, C_{200}$ as this number of isomers produced very smooth looking and confidence-inspiring validation figures. These samples were saved to disk, to avoid having to generate the entire isomerspace every time we want to run a benchmark. The scope of benchmark pipelines and algorithms is limited to the isomerspace range $C_{20}, \cdots, C_{200}$ because generating entire isomerspaces becomes increasingly expensive, the isomerspace sampling of $C_{20}, \cdots, C_{200}$ takes approximately 24 hours.

Information about the exact system specifications and the compilers used in this thesis can be found in Appendix A.

3.4 Parallel Primitives

In parallel programming, there are several building blocks sometimes referred to as *parallel primitives* or *collective operations*. Parallel primitives include, but are not limited to, broadcasting, barrier, gather, scatter, scans and reduction. As these primitives are extremely common efficient realizations of these primitives are of great interest. Specifically, the reduction operation and exclusive scan operation are not implemented in the standard CUDA API as such we take care to implement these efficiently. [1]

3.4.1 Reduction

Algorithm 2 Sequential Reduction

```
function REDUCE(I, ⊕)                          ▷ N input array, associative operator
    S ← 0                                      ▷ Accumulator
    for i ∈ {0, ... N − 1} do
        S ← S ⊕ I[i]
    end for
    return S                                   ▷ Result
end function
```

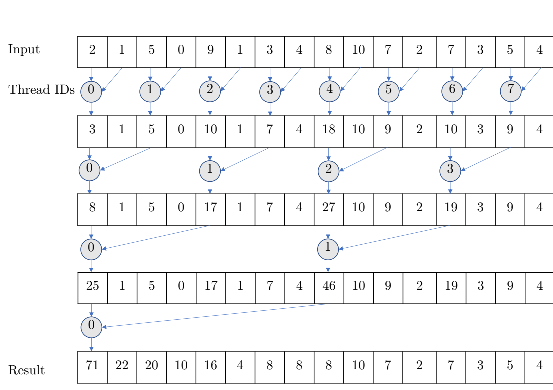
The reduction operation takes a number of elements and reduce them into a single result using any binary and associative operator e.g. addition, subtraction, max, min and others. The sum of a series is one particularly common problem which can be solved with a reduction that is the sequential series of additions $((((((a+b)+c)+d)+e)+f)+g)+h$ can just as well be calculated in parallel $(((a+b)+(c+d))+((e+f)+(g+h)))$. As reduction is essentially an associative operator carried out on a binary tree, runtime complexity of a parallel reduction is $\mathcal{O}(N log_2(N))$. Indeed, we may naively produce a sum in a tree like fashion as depicted in Fig. 3.12a and written in pseudocode in Algorithm 3.

Algorithm 3 In-Place strided parallel reduction

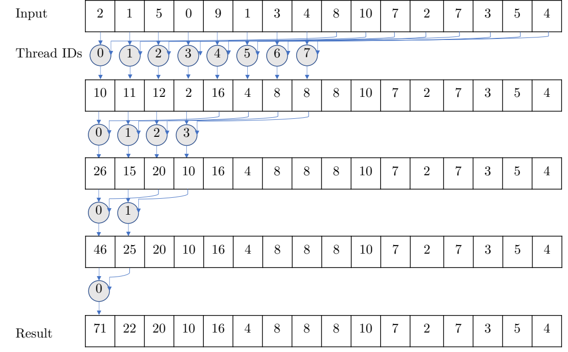
```
function REDUCE(I, ⊕)                          ▷ N × 1 input array, associative operator
    for i ← 1 to ⌈log2(N)⌉ do
        for j ∈ {0, ... 2^(⌈log2(N)⌉−i)} do in lockstep
            I[j ∗ 2^i] ← I[j ∗ 2^i] ⊕ I[j ∗ 2^i + j ∗ 2^(i−1)]
        end for
    end for
    return I[0]                                ▷ Result
end function
```

It turns out however that this approach runs into *memory bank conflicts*. The L1 Cache in Nvidia GPUs is made up of b (typically 32) memory banks that can be accessed simultaneously thus yielding b times the memory bandwidth of a single bank. However, if multiple threads request memory from the same bank the access has to be serialized.

[1] Library implementations from `CUB` exist but require lots of template parameters to be known at compile time



(a) *Tree-like parallel reduction.*



(b) *Contiguous memory access- parallel reduction.*

Figure 3.12: *Two different approaches to parallel reductions.*

For this reason, the tree-like reduction which accesses memory in strides of $2, 4, \ldots 2^N$ will cause $2, 4, \cdots, 2^N$-way bank conflicts. We can remedy this problem by simply addressing sequentially as per Fig. 3.12b. This transformation equates to a change in indexing as seen in Algorithm 4. We can optimize the reduction method one step further by reducing every 32 elements on an input array in the first step using *warp-primitives* and then using a single warp to reduce the resulting elements. The advantage of warp-level primitives is that threads within a warp can exchange data synchronously without going through L1 Cache / Shared memory thus freeing up bandwidth and reducing synchronization overhead we outline the process in Fig. 3.14a.

Algorithm 4 In-Place contiguous parallel reduction

function REDUCE($\boldsymbol{I}, \oplus$) ▷ $N \times 1$ input array and associative operator
  for $i \leftarrow 1$ **to** $\lceil log2(N) \rceil$ **do**
    for $j \in \{0, \ldots 2^{\lceil log2(N) \rceil - i}\}$ **do in lockstep**
      $\boldsymbol{I}[j] \leftarrow \boldsymbol{I}[j] \oplus \boldsymbol{I}[j + 2^{\lceil log2(N) \rceil - i}]$
    end for
  end for
  return $\boldsymbol{I}[0]$ ▷ Result
end function

We proceed to measure the performance of each reduction method by solving as many block-wide reductions as will fit on the GPU at a given block-size (array-size). We run these reductions 1000 times at each array-size and repeat the experiment 10 times to derive standard deviation on the benchmarks. The results are shown in Fig. 3.13

We see that for very small block-sizes the choice of reduction method is unlikely to be vital to performance, however as block-size grows so too does the discrepancy between different algorithms. Already at block-size 200 there is a threefold speedup from using either warp-primitive or contiguous reduction over the sequential method.

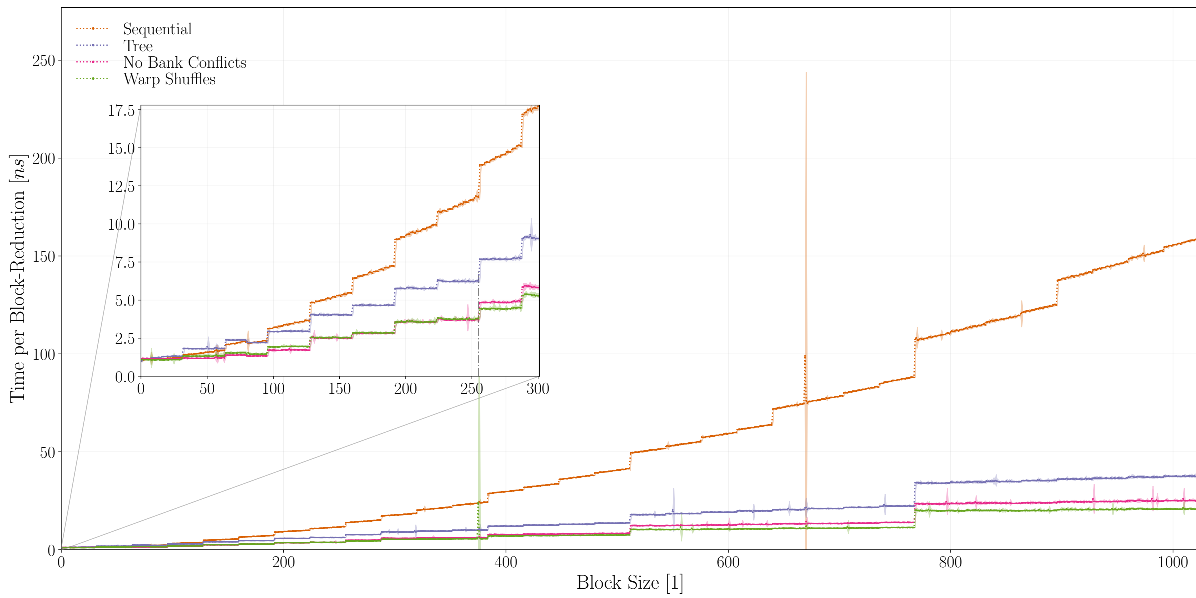
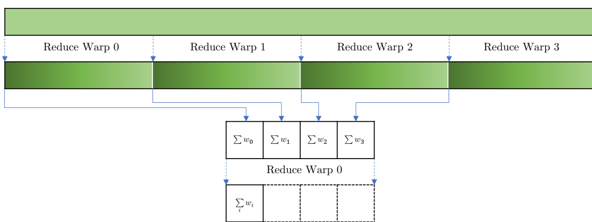
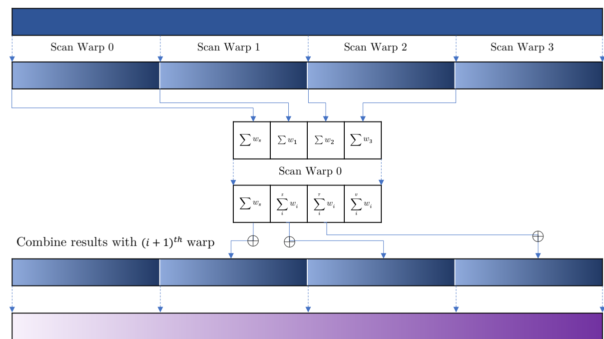


Figure 3.13: *Time per block reduction as a function of block-size. The sequential block reduction in **orange**, tree-like parallel reduction **blue**, contiguous parallel reduction **pink** and warp based divide and conquer reduction **green***





(a) *Warp-first reduction* **(b)** *Warp-first scan*

Figure 3.14: *Visualisation of warp-level reduction and scan operations, the gradient colours represent the relative values of the elements in the array. Darker colours represent larger values. Uniform colour implies the elements are unordered.*

3.4.2 Scan

If the associative operator is addition then the scan operation is the same as a prefix sum, that is the result of applying the scan operation to an array $\boldsymbol{G}$ is:

$$scan(\boldsymbol{G}) = \{A_0, \sum_i^1 A_i, \sum_i^1 A_i, \cdots, \sum_i^{N-1} A_i\} \tag{3.4}$$

The pseudocode for a sequential implementation of the prefix sum is shown in Algorithm 5.

Algorithm 5 Sequential Inclusive Scan

function INCLUSIVESCAN($\boldsymbol{G}, \oplus$) ▷ $N \times 1$ input array
  $\boldsymbol{R}[0] \leftarrow \boldsymbol{G}[0]$
  for $i \leftarrow 1$ **to** $N-1$ **do**
    $\boldsymbol{R}[i] \leftarrow \boldsymbol{R}[i-1] \oplus \boldsymbol{G}[i]$
  end for
  return $\boldsymbol{R}$ ▷ Result
end function

The scan operation is slightly harder to parallelise due to the higher degree of dependency between elements in the array, it is however, possible. One such parallel scan algorithm is the Blelloch which has both the best work-efficiency and time-complexity.[4] This algorithm has been implemented by developers at Nvidia and its performance was measured. The algorithm was in this work, modified slightly. We require the scans to be block-wide instead of grid-wide and furthermore we need to allow for arrays that are not powers of 2 in size. In addition to the modified block-wide Blelloch scan we also implement a divide and conquer approach making use of the API built-in warp-primitive `cooperative_groups::scan()`. The approach is visualized in Fig. 3.14b, and a pseudocode version is presented in Algorithm 6.

The idea is to divide the array into n_w warp-sized sections and scan these first, then we scan the sums of each of these subsections, this corresponds to the ends of each sub-scan, using a single warp and store it in $\boldsymbol{R_2}$. Finally, we can add the ith element $\boldsymbol{R_2}$ to the corresponding i^{th} *section* of $\boldsymbol{R_1}$ and store it back in $\boldsymbol{I}$.

This approach, while both being simple and having worse work-efficiency, turns out to be by far the most efficient, from a hardware perspective this is not surprising, the Blelloch scan accesses memory in a highly non-contiguous fashion as well as requiring intra-block synchronisation at each step of the algorithm. The divide and conquer approach, on the other hand, uses warp-primitives which require no intra-block synchronization and implicitly makes use of the ability for threads within a warp to access each-others thread local memory (registers). We benchmarked the performance of these scan algorithms by performing as many concurrent block scans as the GPU could support 1000 times, looping occurs inside the kernel to produce measurable run-time, standard-deviations were gathered by running this test 10 times for each array size. Results are shown in Fig. 3.15. We see that while Blelloch scan initially does not manage

Algorithm 6 In-Place Divide and Conquer Inclusive Scan

function INCLUSIVESCAN($\boldsymbol{I}, \oplus$) ▷
  $n_w = \lceil \frac{N}{w_s} \rceil$ ▷ Number of subdivisions (warps in CUDA)
  $\boldsymbol{R_1}$ ▷ $n_w \times w_s$ 1st pass scan results
  $\boldsymbol{R_2}$ ▷ $n_w \times 1$ 2nd pass scan results
  $\boldsymbol{S}$ ▷ $n_w \times 1$ Sums of the subsections
  for $i \in \{0, \dots n_w - 1\}$ **do in lockstep**
    ▷ Divide the array into n_w components and scan these
    $\boldsymbol{R_1}[i] = \mathsf{scan}(\boldsymbol{I}[i * w_s, \cdots, (i+1) * w_s - 1], \oplus)$
  end for
  for $i \in \{0, \dots w_s - 1\}$ **do in lockstep**
    $\boldsymbol{S}[i] = \boldsymbol{R_1}[(i+1) * n_w]$ ▷ Stores the ends of each sub-scan contiguously in $\boldsymbol{S}$
  end for
  $\boldsymbol{R_2} = \mathsf{scan}(\boldsymbol{S}[0], \boldsymbol{S}[w_s - 1])$ ▷ Scan the result of the sub-scans
  for $i \in \{0, \dots n_w - 1\}$ **do in lockstep**
    for $j \in \{0, \dots w_s - 1\}$ **do in lockstep**
      $\boldsymbol{I}[i * n_w + j] = \boldsymbol{R_2}[i] \oplus \boldsymbol{R_1}[i][j]$
    end for
  end for
end function

to outperform the very simple sequential, yet inter-block parallel, scan, it does starts to win out at around block-size 300. Given that the current primary domain of interest are isomerspaces $C_{20}, C_{24}, \cdots, C_{300}$ the development of the divide and conquer approach really pays off as the per block-scan is an order of magnitude faster at block-size 250. We shall see later how the efficient divide and conquer algorithm enables an exceedingly performant dualisation algorithm.

3.5 Summary

In Section 3.1 we have discussed the basics of GPGPU compute, memory and software hierarchy (Fig. 3.2), given a brief introduction to CUDA and some of the terminology that follows. We go through occupancy, and *effective occupancy* (Fig. 3.4) and evaluate in great detail how the occupancy of a kernel is affected by block-size and registers per thread (Fig. 3.6, Fig. 3.5, Fig. 3.7). We find that only a small subset of compilation settings for the `-maxrregcount` flag, actually affect occupancy. Furthermore, we saw that 128 is a perfect block size for evaluating kernel performance for different RPT `-maxrregcount` settings (Fig. 3.6). In Section 3.1.4 we discussed the effect of throttling on performance and how we need to accommodate for this by *warming* the GPUs up prior to benchmarks (Fig. 3.9).

In Section 3.3 we outlined how we intend to ensure that isomers used for benchmarks and validity testing actually are representative of the isomerspaces they belong to, by creating and storing uniformly distributed isomers from each isomerspace. The approach was described in Algorithm 1.

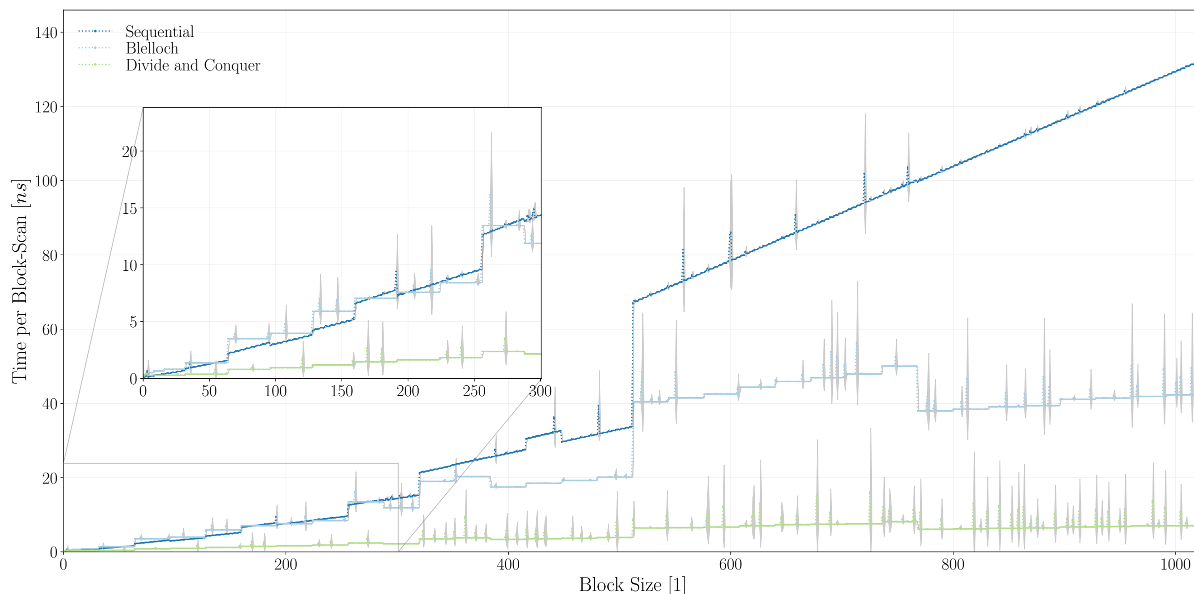


Figure 3.15: *Time per Block scan using the sequential (**blue**) algorithm, Blelloch scan (**light blue**) and the divide and conquer approach (**green**). Each measurement is derived from 10 runs of 1000 intra-kernel executions. We test the performance at all the possible array sizes up to a maximum of 1024 elements i.e. the block-size limit.*

In Section 3.4 we have discussed how operations such as scans and reductions, can be turned into highly performant parallel primitives on GPUs. Specifically we have seen the way bank conflicts affects performance as well as the effect that using warp-primitives and a divide and conquer approach can improve performance in spite of higher work load for both scan and reduction primitives. We found that using divide and conquer warp-primitive scan approach improves performance an order of magnitude over a sequential scan at $N = 200$ and even outperforms the theoretically best scaling, Blelloch scan. We shall use these primitives in numerous algorithms in this thesis and as such they play a central role.

Chapter 4

Design and Implementation of Lockstep Parallel Forcefield Algorithm

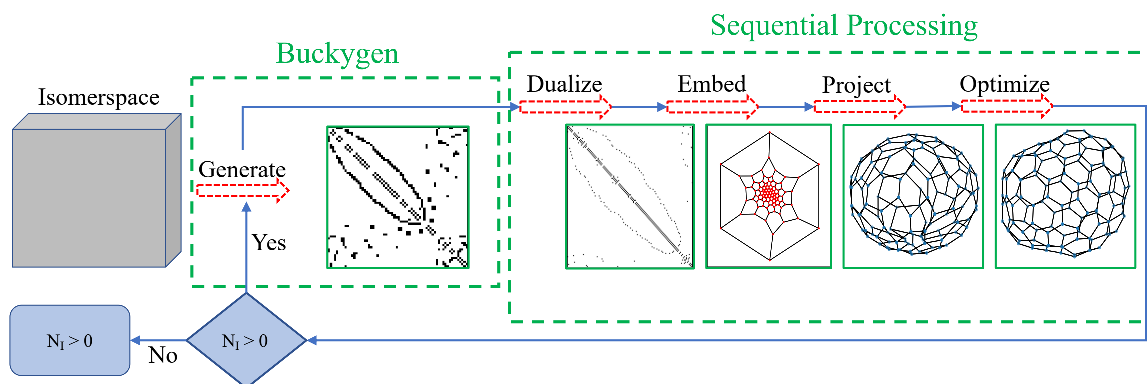


Figure 4.1: *Sequential pipeline for molecular geometry optimisation. Blue arrows indicate the flow of control while red arrows depict operations performed on data. Green boxes display the data residing on the CPU.*

4.1 Overview and Motivation

Let us for a moment consider the steps required for producing an optimal molecular geometry for a single molecule. We depict a combined flow-chart and state diagram of such a system in Fig. 4.1. The sequential pipeline is as follows: we produce a face graph using *BuckyGen* then **dualize** the graph such that the vertices denote atoms of the system, penultimately embed the graph using **Tutte embedding** and **project** the geometry on to a sphere to produce. Finally, we use a forcefield method to **optimize** the geometry of the molecule. The full script for this pipeline implementation can be found in Listing C.1. More details and exposition will be given on the Dualisation, Embedding and Projection procedures in Chapter 5.

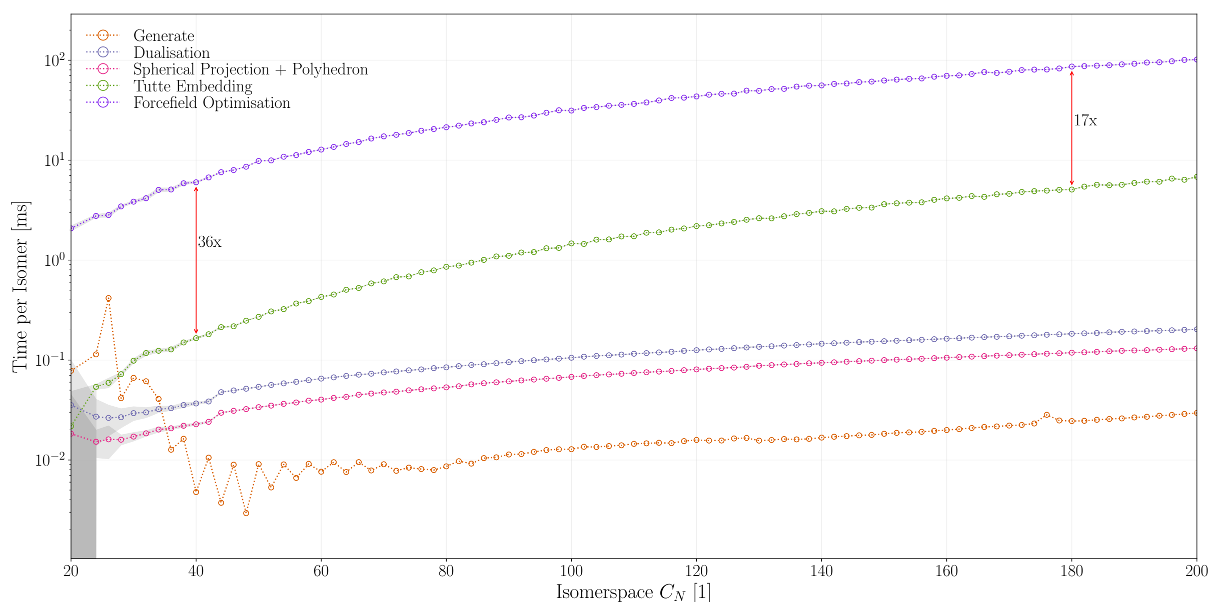


Figure 4.2: *Time per isomer of pipeline components benchmarked for isomerspaces $C_{20}, C_{24}, \cdots, C_{200}$. Note that the y-axis is logarithmic. Note that sequential components are denoted by $\circ$ and parallel components by $\star$.*

Benchmarking the components of this pipeline yielded the results shown in Fig. 4.2. As we can see, the bottleneck of the pipeline is the forcefield optimisation step, at isomerspace C_{190} it takes 17 times longer to optimise than to embed the geometry in 2D. This is not surprising as the forcefield optimisation step is by far the most computationally intensive step in the pipeline. Amdahl's law states that the total application speedup that one can achieve from parallelisation is limited by the fraction of the application which is sequential, see Eq. (4.1).

$$S(s) = \frac{1}{1 - p + \frac{p}{s}} \tag{4.1}$$

p denotes the fraction of the application runtime which benefits from parallelisation, s is the speedup of the parallel component. In our case if we initially assume that the forcefield optimisation is the only parallelisable component, then for C_{200}, $p = 0.956$ which means in the limit $\lim_{s\to\infty}$ we can expect a speedup of $\frac{1}{1-0.956} = 21.7\times$.

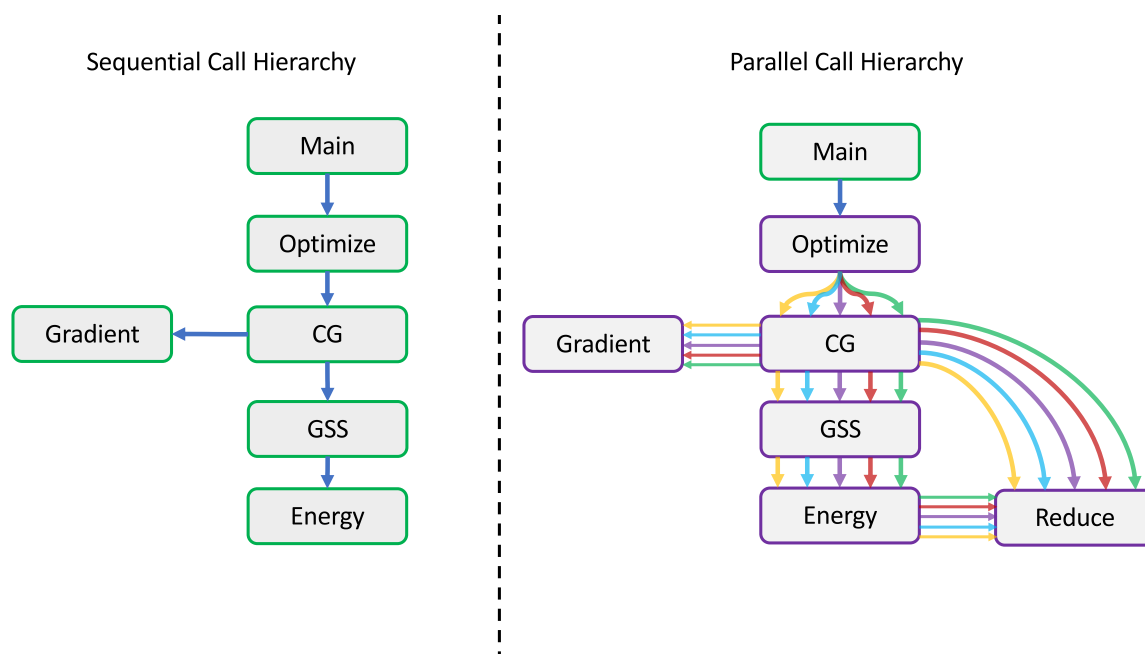


Figure 4.4: *The sequential calling hierarchy is shown to the left of the figure. The parallel calling hierarchy is shown to the right. Green boxes signify computations performed on the CPU while purple boxes denote computations performed on the GPU. The coloured arrows in the parallel calling hierarchy each represent a collection of threads responsible for the computations of a single isomer.*

4.2 First Design

Given that we would be able to derive this much performance from parallelising the forcefield optimisation let us break down the forcefield optimisation problem into its constituents. The calling hierarchy is shown in Fig. 4.4. The hierarchy consists of a main thread which calls optimize, in the sequential case a single thread then optimises the geometry using conjugate gradient descent coupled with golden section line search (GSS). Each of these functions in turn call functions to evaluate the gradient and energy of the system. Before we can discuss these components we must first present some prerequisite utility functions.

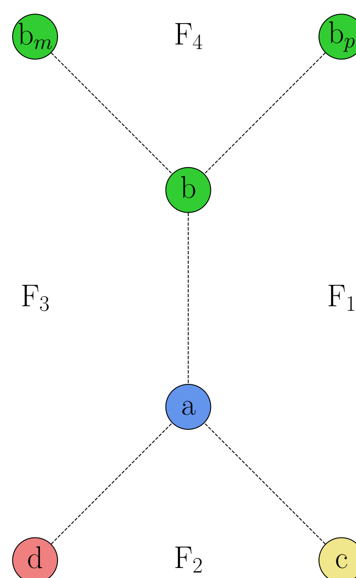


Figure 4.3: *Labelling of the faces of the graph from the perspective of the arc (* $\boldsymbol{a} \rightarrow \boldsymbol{b}$ *).*

4.2.1 *Utility Functions*

In subsequent algorithms in this section and the next, we will require methods for accessing and traversing the graph structure and inquiring about neighbouring faces. In Algorithm 7 we define these functions: `ArcIdx`, returns the index of a neighbour in the neighbour list of a vertex. `Next`

takes a vertex u and a neighbour v and returns the next neighbour of that vertex. `Prev` returns the previous neighbour. `NextOnFace` and `PrevOnFace` return the next and previous neighbours on a face given an arc. `FaceSize` returns the size of the face associated with an arc. These algorithms rely on the key fact that the neighbour list $\boldsymbol{G}$ has a canonical ordering of neighbours, i.e. the neighbours are ordered in a clockwise fashion around the vertex. And we are therefore able to associate any arc with a face: using labelling found in figure Fig. 4.3 the arc $a \to b$ corresponds to F_1, $a \to c$ to F_2, $b \to b_m$ gives F_4 and so on.

The utility methods just presented existed already in the *fullerene* program, but had to be implemented in the CUDA code as well such that these methods could be computed in parallel on the GPU, this implementation can be found in Listing B.1. Granted these tools we are now ready to compute the constants required for the energy and gradient functions. While both the *fullerene* program provided this in one form and Pedersen implemented methods to compute these constants, neither are clearly documented, and so we provide in Algorithm 8 pseudocode for this, the method is also implemented in CUDA and can be found in Listing B.2.

Algorithm 7 Utility Functions

```
1: function ArcIdx(G, u, v)
2:     for j ∈ {0, 1, 2} do                                   ▷ For each neighbour
3:         if G[u, j] = v then return j ▷ Return the index of v in the neighbour list of u
4:         end if
5:     end for
6: end function
7:
8: function Next(G, u, v)     ▷ Given a canonical edge and a graph, returns the next
   neighbour
9:     j = ArcIdx(G, u, v)                      ▷ Get the index of the current neighbour
10:     return G[u, (j + 1) mod 3]    ▷ The next neighbour is the neighbour one index
   after the current neighbour
11: end function
12:
13: function Prev(G, u, v)       ▷ Given the arc {u, v}, returns the previous neighbour
14:     j = ArcIdx(G, u, v)
15:     return G[u, (j + 2) mod 3]      ▷ The previous neighbour is the neighbour two
   indices after the current neighbour
16: end function
17:
18: ▷ Given the arc {u, v}, returns the next neighbour on the face
19: function NextOnFace(G, u, v) return prev(v, u)
20: end function
21:
22: ▷ Given the arc {u, v}, returns the previous neighbour on the face
23: function PrevOnFace(G, u, v) return next(v, u)
24: end function
25:
26: function FaceSize(G, u, v)                ▷ Given the arc {u, v}, returns size of the face
27:     d = 1                                        ▷ Initialise the size of the face to 1
28:     u_0 = u                                                    ▷ Store the source vertex
29:     while v ≠ u_0 do              ▷ While we have not returned to the source vertex
30:         w = v                                                 ▷ Store the current vertex
31:         v = NextOnFace(u, v)                          ▷ Get the next vertex on the face
32:         u = w                                            ▷ Set u to be the previous v
33:         d = d + 1                                       ▷ Increment the size of the face
34:     end whilereturn d
35: end function
36: function GetFace(G, u, v)      ▷ Given the arc {u, v}, returns the face and its size
37:     F = {∞, ∞, ∞, ∞, ∞, ∞}                    ▷ Initialise the face to be a vector of size 6
38:     F[0] = u               ▷ Set the first element of the face to be the source vertex
39:     d = 1                                        ▷ Initialise the size of the face to 1
40:     while v ≠ F[0] do             ▷ While we have not returned to the source vertex
41:         w = v                                  ▷ Temp variable to store the current v
42:         v = NextOnFace(u, v)                          ▷ Get the next vertex on the face
43:         u = w                                            ▷ Set u to be the previous v
44:         F[d] = v                                       ▷ Add the new vertex to the face
45:         d = d + 1                                       ▷ Increment the size of the face
46:     end while
47:     return F
48: end function
```

Algorithm 8 Constants

1: **function** CONSTANTS($\boldsymbol{G}$)
2: $r_0 \leftarrow [[R_{pp}, R_{ph}], [R_{hp}, R_{hh}]]$
3: $\theta_0 \leftarrow [\theta_p, \theta_h]$ ▷ Equilibrium angles
4: $\phi_0 \leftarrow [[[\phi_{ppp}, \phi_{pph}], [\phi_{php}, \phi_{phh}]], [[\phi_{hpp}, \phi_{hph}], [\phi_{hhp}, \phi_{hhh}]]]$
5: $K_{bond} \leftarrow [[K_{pp}, K_{ph}], [K_{hp}, K_{hh}]]$ ▷ Bond constants
6: $K_{ang} \leftarrow [K_p, K_h]$ ▷ Angle constants
7: $K_{dih} \leftarrow [[[K_{ppp}, K_{pph}], [K_{php}, K_{phh}]], [[K_{hpp}, K_{hph}], [K_{hhp}, K_{hhh}]]]$
8: $\boldsymbol{R_0}, \boldsymbol{\Theta_0}, \boldsymbol{\phi_0}, \boldsymbol{\Theta_{m_0}}, \boldsymbol{\Theta_{p_0}}, \boldsymbol{\phi_{a_0}}, \boldsymbol{\phi_{m_0}}, \boldsymbol{\phi_{p_0}}$ ▷ Arrays of constants
9: $\boldsymbol{K_{bond}}, \boldsymbol{K_{ang}}, \boldsymbol{K_{dih}}, \boldsymbol{K_{ang_m}}, \boldsymbol{K_{ang_p}}, \boldsymbol{K_{dih_a}}, \boldsymbol{K_{dih_m}}, \boldsymbol{K_{dih_p}}$ ▷ Arrays of constants
10: **for** $i \in \{0, \ldots, N-1\}$ **do** ▷ For each atom
11: **for** $j \in \{0, 1, 2\}$ **do** ▷ For each neighbour
12: $F_1 \leftarrow$ FaceSize($\boldsymbol{G}, i, \boldsymbol{G}[i, j]$) $- 5$
13: $F_2 \leftarrow$ FaceSize($\boldsymbol{G}, i, \boldsymbol{G}[i, j+1 \mod 3]$) $- 5$
14: $F_3 \leftarrow$ FaceSize($\boldsymbol{G}, i, \boldsymbol{G}[i, j+2 \mod 3]$) $- 5$
15: $F_4 \leftarrow$ FaceSize($\boldsymbol{G}, \boldsymbol{G}[i, j]$, PrevOnFace($\boldsymbol{G}[i, j], i$)) $- 5$
16: $\boldsymbol{R_0}[i, j] \leftarrow r_0[F1, F2]$
17: $\boldsymbol{K_{bond}}[i, j] \leftarrow K_{bond}[F1, F2]$
18: $\boldsymbol{\Theta_0}[i, j] \leftarrow \theta_0[F1]$
19: $\boldsymbol{K_{ang}}[i, j] \leftarrow K_{ang}[F1]$
20: $\boldsymbol{\phi_0}[i, j] \leftarrow \phi_0[F1, F2, F3]$
21: $\boldsymbol{K_{dih}}[i, j] \leftarrow K_{dih}[F1, F2, F3]$
22: $\boldsymbol{\Theta_{m_0}}[i, j] \leftarrow \theta_0[F3]$
23: $\boldsymbol{K_{ang_m}}[i, j] \leftarrow K_{ang}[F3]$
24: $\boldsymbol{\Theta_{p_0}}[i, j] \leftarrow \theta_0[F1]$
25: $\boldsymbol{K_{ang_p}}[i, j] \leftarrow K_{ang}[F1]$
26: $\boldsymbol{\phi_{a_0}}[i, j] \leftarrow \phi_0[F3, F4, F1]$
27: $\boldsymbol{K_{dih_a}}[i, j] \leftarrow K_{dih}[F3, F4, F1]$
28: $\boldsymbol{\phi_{m_0}}[i, j] \leftarrow \phi_0[F4, F1, F3]$
29: $\boldsymbol{K_{dih_m}}[i, j] \leftarrow K_{dih}[F4, F1, F3]$
30: $\boldsymbol{\phi_{p_0}}[i, j] \leftarrow \phi_0[F1, F3, F4]$
31: $\boldsymbol{K_{dih_p}}[i, j] \leftarrow K_{dih}[F1, F3, F4]$
32: **end for**
33: **end for**
34: **return** $\{\boldsymbol{R_0}, \boldsymbol{\Theta_0}, \boldsymbol{\phi_0}, \boldsymbol{\Theta_{m_0}}, \boldsymbol{\Theta_{p_0}}, \boldsymbol{\phi_{a_0}}, \boldsymbol{\phi_{m_0}}, \boldsymbol{\phi_{p_0}},$
35: $\boldsymbol{K_{bond}}, \boldsymbol{K_{ang}}, \boldsymbol{K_{dih}}, \boldsymbol{K_{ang_m}}, \boldsymbol{K_{ang_p}}, \boldsymbol{K_{dih_a}}, \boldsymbol{K_{dih_m}}, \boldsymbol{K_{dih_p}}\}$
36: **end function**

Lines [2 - 7] in Algorithm 8 represent the forcefield parameters discussed in Section 2.2.1 they are intended to be indexed with face sizes, 0 for pentagons, 1 for hexagons. For instance $K_{bond}[0,0]$ is the bond force constant for a pentagon-pentagon bond, $K_{dih}[1,1,1]$ is the dihedral force constant for a dihedral angle in which the faces involved are hexagon-hexagon-hexagon. The bold face constants, lines [8 - 9] are arrays of constants each storing 1 constant for each of the 3 neighbours of each atom. Fig. 4.3 shows the faces involved in Algorithm 8 from the perspective of the arc (**a** $\rightarrow$ **b**), for arc (**a** $\rightarrow$ **c**) F_1 becomes F_3, F_2 becomes F_1 and F_3 becomes F_1. We can realize from Algorithm 8 that in the ith iteration only the ith elements of the constant parameters are written to,

thus no loop carried dependencies are present, and the process is entirely parallelisable. Having presented a way of computing these constants we can now discuss conjugated gradient descent.

4.2.2 Conjugate Gradient Descent

Treating the geometry of the fullerene as an energy minimisation problem, we require an optimisation algorithm that finds a good balance between rate of convergence and computational cost per step. One might consider the newton method, which has the desirable property: quadratic convergence. This method, however, requires the computation and storage of the Hessian matrix which often is prohibitively expensive. In our case this is a $3N \times 3N$ matrix.

A number of quasi-newton methods exist which do not require the computation of the Hessian yet continuously updates an approximation of the hessian matrix. The most popular of these methods is the *Broyden-Fletcher-Goldfarb-Shanno* (BFGS) method. The trouble of course being that we now actually need to store this approximate Hessian and compute matrix-vector multiplications.

Limited-memory BFGS (L-BFGS) is a variant of the BFGS method which only stores m updates to the position $\boldsymbol{X}$ and the gradient $\boldsymbol{\nabla} f(\boldsymbol{X})$ as these are then used to implicitly perform operations that otherwise would have been matrix-vector products. The L-BFGS is certainly a candidate for our problem, however, we wish to also be able to compare our results to the implementation of Wirz et al.[18]

Enter the *Conjugate Gradient* (CG) method, this is the method used by Wirz et al.[18] and Pedersen.[12] CG was shown to be equivalent to the L-BFGS method with $m = 0$.[13] Indeed, Nazareth compared the performance of BFGS to that of CG noting that one typically has to perform twice as many iterations to achieve the same accuracy with CG as with BFGS. However, CG is much cheaper to compute, and it does not require the storage of the Hessian matrix. In general CG requires N iterations to converge for a system of N equations, however our system is nonlinear and therefore convergence may be slower, more on that in Section 4.4.2. As a side note, it is possible to show that for our system, the sparsity of this Hessian is quite high, thus it may be a possible avenue to explore in the future, although a lot of work is required to find the analytical second derivatives and implement it efficiently.

In Algorithm 9 we present the classical conjugate gradient descent algorithm, with the distinction that all linear algebra operations are shown as explicit for loops iterating over data. Furthermore, the gradient direction at the previous and current iteration is explicitly denoted $\boldsymbol{g}_t$ and $\boldsymbol{g}_{t+1}$. The purpose of these design choices is that we wish to be very explicit about the data storage and dependencies of the algorithm and to make clear which parts can be parallelised and which variables are shared between threads of a block, which are required across the device and which are private.

Algorithm 9 Sequential Conjugate Gradient Optimisation

```
1:  function CONJUGATE GRADIENT(X_in, G_in)
2:      for i ∈ {0, ..., M − 1} do                                  ▷ For each isomer
3:          for a ∈ {0, ..., N − 1} do
4:              X[a] ← X_in[i, a]
5:              G[a] ← G_in[i, a]
6:          end for
7:          C ← Constants(G)                        ▷ Multiple N × 3 arrays of constants
8:          g_t                        ▷ N × 3 array of gradients at previous iteration
9:          g_{t+1}                     ▷ N × 3 array of gradients at current iteration
10:         s                                    ▷ N × 3 array of search directions
11:         for a ∈ {0, ..., N − 1} do                                 ▷ For each atom
12:             g_t[a] ← gradient(X, C, a)          ▷ Compute the gradient w.r.t atom a
13:             s[a] ← −g_t[a]                                    ▷ Search direction
14:         end for
15:         s_n ← 0                                ▷ Norm of search direction squared
16:         for a ∈ {0, ..., N − 1} do                                 ▷ For each atom
17:             s_n ← s_n + s[a] · s[a]        ▷ Compute the norm of the search direction
18:         end for
19:         for a ∈ {0, ..., N − 1} do                                 ▷ For each atom
20:             s[a] ← s[a] / √s_n                    ▷ Normalize the search direction
21:         end for
22:         for t ∈ {0, ..., 5N} do                             ▷ Perform 5N iterations
23:             α ← GSS(X, s, X1, X2)                         ▷ Perform a line search
24:             for a ∈ {0, ..., N − 1} do                             ▷ For each atom
25:                 g_{t+1}[a] ← gradient(X, C, a)  ▷ Compute the gradient w.r.t atom a
26:             end for
27:             β ← 0                                   ▷ Polak-Ribiere coefficient
28:             for a ∈ {0, ..., N − 1} do                             ▷ For each atom
29:                 β ← β + (g_{t+1}[a] − g_t[a]) · g_{t+1}[a]
30:                 X[a] ← X[a] + αs[a]                     ▷ Update the coordinates
31:             end for
32:             for a ∈ {0, ..., N − 1} do                             ▷ For each atom
33:                 s[a] ← −g_{t+1}[a] + βs[a]            ▷ Update the search direction
34:                 g_t[a] ← g_{t+1}[a]    ▷ Assign the new gradient to the old gradient
35:             end for
36:             s_n ← 0                            ▷ Norm of search direction squared
37:             for a ∈ {0, ..., N − 1} do                             ▷ For each atom
38:                 s_n ← s_n + s[a] · s[a]    ▷ Compute the norm of the search direction
39:             end for
40:             for a ∈ {0, ..., N − 1} do                             ▷ For each atom
41:                 s[a] ← s[a] / √s_n                ▷ Normalize the search direction
42:             end for
43:             for a ∈ {0, ..., N − 1} do                             ▷ For each atom
44:                 X[a] ← X[a] + αs[a]                     ▷ Update the coordinates
45:             end for
46:         end for
47:     end for
48: end function
```

Naturally, the outer loop over isomers has no dependency, this is task-level parallelism, and it follows that if the contents of each iteration of this loop perform the same operations, then it is suitable for lockstep parallelism. Let us now inspect the contents of this loop.

As per previous, the argument of no loop carried dependencies applies in Algorithm 9 as well. We have intentionally designed the gradient function such that it returns the gradient of a single atom. This allows us to parallelise the gradient calculation over all atoms in the molecule. Notice that some for loops over atoms ($a \in \{0, \ldots, N-1\}$) in Algorithm 9, require the result of a previous iteration. Fortunately, these loops are all a matter of summation and so while they are not perfectly parallelisable we can use parallel reductions to compute these sums in $\mathcal{O}(log_2(N)N)$ time. If we can show that *Golden Section Search* is a collective operation that is computable in parallel, then Algorithm 10 constitutes a parallel Conjugated Gradient Descent algorithm.

Algorithm 10 Parallel Conjugate Gradient Descent

```
1: function CONJUGATE GRADIENT(X_in, G_in)                      ▷ Initial coordinates, Graphs
2:     for i ∈ {0, ..., M − 1} do in lockstep                    ▷ For each isomer
3:         X ← X_in[i]                                           ▷ Copy the coordinates
4:         G ← G_in[i]                                           ▷ Copy the neighbour list
5:         C ← Constants(G)                                      ▷ Multiple N × 3 arrays of constants
6:         g_t                                                   ▷ N × 3 array of gradients at previous iteration
7:         g_{t+1}                                               ▷ N × 3 array of gradients at current iteration
8:         s                                                     ▷ N × 3 array of search directions
9:         for a ∈ {0, ..., N − 1} do in lockstep                ▷ For each atom
10:            g_t[a] ← gradient(X, G, C, a)                     ▷ ∂E/∂X[a]
11:            s[a] ← −g_t[a]                                    ▷ Search direction
12:        end for
13:        s_n ← reduce(s[a] · s[a], +)                          ▷ Norm of search direction squared
14:        for a ∈ {0, ..., N − 1} do in lockstep                ▷ For each atom
15:            s[a] ← s[a] / √s_n                                ▷ Normalize the search direction
16:        end for
17:        for t ∈ {0, ..., 5N − 1} do                           ▷ Perform 5N conjugate gradient steps
18:            α ← GoldenSectionSearch(X, G, C, s)               ▷ Find best step size along s
19:            for a ∈ {0, ..., N − 1} do in lockstep            ▷ For each atom
20:                g_{t+1}[a] ← gradient(X, G, C, a)             ▷ ∂E/∂X[a]
21:            end for
22:            β_top ← reduce((g_{t+1}[a] − g_t[a]) · g_{t+1}[a], +)   ▷ Denominator of
    Polak-Ribiere
23:            β_bottom ← reduce(g_t[a] · g_t[a], +)             ▷ Numerator of Polak-Ribiere
24:            β ← β_top/β_bottom                                ▷ Polak-Ribiere
25:            for a ∈ {0, ..., N − 1} do in lockstep            ▷ For each atom
26:                X[a] ← X[a] + αs[a]                           ▷ Update the coordinates
27:            end for
28:            for a ∈ {0, ..., N − 1} do in lockstep            ▷ For each atom
29:                s[a] ← −g_{t+1}[a] + βs[a]                    ▷ Update the search direction
30:                g_t[a] ← g_{t+1}[a]                           ▷ Update the gradient
31:            end for
32:            s_n ← √reduce(s[a] · s[a], +)                     ▷ Norm of search direction
33:            for a ∈ {0, ..., N − 1} do in lockstep            ▷ For each atom
34:                s[a] ← s[a]/s_n                               ▷ Normalize the search direction
35:            end for
36:        end for
37:        X_in[i] ← X                                           ▷ Store the result
38:    end for
39: end function
```

In our parallel algorithms we specify the storage location of each variable through colours. We use $\boldsymbol{X_{in}}$ to signify that a variable is a global variable in the sense that it must be accessible to all processing elements on the device. $\boldsymbol{X}$ implies that a variable needs to be accessed across processing elements within a single isomer (sub-problem), this is the case for both the coordinate vector $\boldsymbol{X}$ and $\boldsymbol{G}$ seeing as the ath atom needs

to access the coordinates of its neighbours and similarly needs to fetch the neighbours of its neighbours. g_t specifies that if the number of processing elements matches the number of atoms, then the variable is local and can reside permanently in registers, otherwise, it must be stored in some form of shared memory. Finally, variables without colour are always local regardless of the number of processing elements.

Now let us proceed to discuss the subroutines involved in Algorithm 10. Starting with the gradient subroutine, shown in Algorithm 11.

Algorithm 11 Gradient

1: **function** GRADIENT($\boldsymbol{X}$, $\boldsymbol{G}$, $\boldsymbol{C}$, a)
2: $\boldsymbol{g} \leftarrow \{0, 0, 0\}$
3: **for** $k \in \{0, 1, 2\}$ **do** ▷ For each neighbour
4: $b \leftarrow \boldsymbol{G}[a, k]$ ▷ Neighbour atom
5: $c \leftarrow \boldsymbol{G}[a, (k+1)\%3]$ ▷ $((k+1)\%3)^{th}$ neighbour of atom a
6: $d \leftarrow \boldsymbol{G}[a, (k+2)\%3]$ ▷ $((k+2)\%3)^{th}$ neighbour of atom a
7: $b_p \leftarrow$ next_on_face($\boldsymbol{G}$, a, b) ▷ Next atom on the face represented by the arc (a, b)
8: $b_m \leftarrow$ prev_on_face($\boldsymbol{G}$, a, b) ▷ Previous atom on the face represented by the arc (a, b)
9: $\boldsymbol{ab} \leftarrow \boldsymbol{X}[b] - \boldsymbol{X}[a]$ ▷ Vector from a to b
10: $\boldsymbol{ac} \leftarrow \boldsymbol{X}[c] - \boldsymbol{X}[a]$ ▷ Vector from a to c
11: $\boldsymbol{ad} \leftarrow \boldsymbol{X}[d] - \boldsymbol{X}[a]$ ▷ Vector from a to d
12: $\boldsymbol{ab_p} \leftarrow \boldsymbol{X}[b_p] - \boldsymbol{X}[a]$ ▷ Vector from a to b_p
13: $\boldsymbol{ab_m} \leftarrow \boldsymbol{X}[b_m] - \boldsymbol{X}[a]$ ▷ Vector from a to b_m
14: $\boldsymbol{bb_p} \leftarrow \boldsymbol{X}[b_p] - \boldsymbol{X}[b]$ ▷ Vector from b to b_p
15: $\boldsymbol{bb_m} \leftarrow \boldsymbol{X}[b_m] - \boldsymbol{X}[b]$ ▷ Vector from b to b_m
16: $\boldsymbol{bc} \leftarrow \boldsymbol{X}[c] - \boldsymbol{X}[b]$ ▷ Vector from b to c
17: $\boldsymbol{cd} \leftarrow \boldsymbol{X}[d] - \boldsymbol{X}[c]$ ▷ Vector from c to d
18: $\boldsymbol{b_m b_p} \leftarrow \boldsymbol{X}[b_p] - \boldsymbol{X}[b_m]$ ▷ Vector from b_m to b_p
19: $\boldsymbol{g}_{bond} \leftarrow \boldsymbol{K_{bond}}[a, k] * \widehat{\boldsymbol{ab}} * (\|\boldsymbol{ab}\| - \boldsymbol{R_0}[a, k])$ ▷ Bond gradient
20: $\theta \leftarrow \widehat{\boldsymbol{ab}} \cdot \widehat{\boldsymbol{ac}}$ ▷ Angle between $\boldsymbol{ab}$ and $\boldsymbol{ac}$
21: $\boldsymbol{g}_{ang} \leftarrow \boldsymbol{K_{ang}}[a, k](\frac{\widehat{\boldsymbol{ab}}\theta - \widehat{\boldsymbol{ac}}}{\|\boldsymbol{ab}\|} + \frac{\widehat{\boldsymbol{ac}}\theta - \widehat{\boldsymbol{ab}}}{\|\boldsymbol{ac}\|})(\theta - \boldsymbol{\Theta_0}[a, k])$ ▷ Angle gradient
22: $sin(\theta_{abc}) \leftarrow \sqrt{1 - (-\widehat{\boldsymbol{ab}} \cdot \widehat{\boldsymbol{bc}})^2}$ ▷ Sin of angle between $\boldsymbol{ba}$ and $\boldsymbol{bc}$
23: $sin(\theta_{bcd}) \leftarrow \sqrt{1 - (-\widehat{\boldsymbol{bc}} \cdot \widehat{\boldsymbol{cd}})^2}$ ▷ Sin of angle between $\boldsymbol{bc}$ and $\boldsymbol{cd}$
24: $\widehat{\boldsymbol{n}}_{abc} \leftarrow \frac{-\widehat{\boldsymbol{ab}} \times \widehat{\boldsymbol{bc}}}{sin(\theta_{abc})}$ ▷ Normal to the plane containing $\boldsymbol{ab}$, $\boldsymbol{bc}$ and $\boldsymbol{ac}$
25: $\widehat{\boldsymbol{n}}_{bcd} \leftarrow \frac{-\widehat{\boldsymbol{bc}} \times \widehat{\boldsymbol{cd}}}{sin(\theta_{bcd})}$ ▷ Normal to the plane containing $\boldsymbol{bc}$, $\boldsymbol{cd}$ and $\boldsymbol{bd}$
26: $\phi \leftarrow \widehat{\boldsymbol{n}}_{abc} \cdot \widehat{\boldsymbol{n}}_{bcd}$ ▷ Angle between $\widehat{\boldsymbol{n}}_{abc}$ and $\widehat{\boldsymbol{n}}_{bcd}$
27: $\boldsymbol{g}_{dih} \leftarrow \frac{\widehat{\boldsymbol{bc}} \times \widehat{\boldsymbol{n}}_{bcd}}{sin(\theta_{abc})\|\boldsymbol{ab}\|} + \widehat{\boldsymbol{ab}}\phi + \frac{cot(\theta_{abc})\phi}{\|\boldsymbol{ab}\|}(\widehat{\boldsymbol{bc}} + \widehat{\boldsymbol{ab}}cos(\theta_{abc}))(\phi - \boldsymbol{\Phi_0}[a, k])$
28: $\boldsymbol{g}_{dih_a} \leftarrow \cdots$ ▷ gradient w.r.t. outer dihedral plane $\widehat{\boldsymbol{n}}_{amp}$
29: $\boldsymbol{g}_{dih_m} \leftarrow \cdots$ ▷ gradient w.r.t. outer dihedral plane $\widehat{\boldsymbol{n}}_{mpa}$
30: $\boldsymbol{g}_{dih_p} \leftarrow \cdots$ ▷ gradient w.r.t. outer dihedral plane $\widehat{\boldsymbol{n}}_{pam}$
31: $\boldsymbol{g}_{ang_m} \leftarrow \cdots$ ▷ gradient w.r.t. outer angle between $\widehat{\boldsymbol{ab}}$ and $\widehat{\boldsymbol{bb_m}}$
32: $\boldsymbol{g}_{ang_p} \leftarrow \cdots$ ▷ gradient w.r.t. outer angle between $\widehat{\boldsymbol{ab}}$ and $\widehat{\boldsymbol{bb_p}}$
33: $\boldsymbol{g} \leftarrow \boldsymbol{g} + \boldsymbol{g}_{bond} + \boldsymbol{g}_{ang} + \boldsymbol{g}_{dih} + \boldsymbol{g}_{dih_a} + \boldsymbol{g}_{dih_m} + \boldsymbol{g}_{dih_p} + \boldsymbol{g}_{ang_m} + \boldsymbol{g}_{ang_p}$ ▷ Total gradient
34: **end for**
35: **return** $\boldsymbol{g}$ ▷ Return the gradient
36: **end function**

In Algorithm 11 we demonstrate how the gradient with respect to a single atom can be encapsulated in a function. This function is called for each atom in the system, either sequentially or in parallel, therefore we need not divide this algorithm into a parallel

gradient and a sequential one. We note that $\boldsymbol{G}$ is accessed the same way by the ath call to the function every time, and therefore we might store these neighbour values ($\boldsymbol{G}[a,0]$, $\boldsymbol{G}[a,1]$, $\boldsymbol{G}[a,2]$), as well as next_on_face($\boldsymbol{G}[a,k]$) and prev_on_face($\boldsymbol{G}[a,k]$) $\forall k \in [0,2]$ in thread private memory. Thus, further reducing the number of memory accesses and increasing the performance. The implementation details can be found in Listing B.8.

While linear conjugate gradient descent allows the direct closed-form solution to the optimal step length α at each iteration, nonlinear conjugate gradient descent requires us to find the optimal step length α using a line search algorithm. We use the sequential GSS to find the optimal step length. Since it only requires us to evaluate the objective function (energy) rather than the gradient.

Algorithm 12 Sequential Golden Section Search

```
function GoldenSectionSearch(X, G, C, d)
    X_1                                         ▷ N × 3 System coordinates at x_1
    X_2                                         ▷ N × 3 System coordinates at x_2
    a ← 0                                       ▷ Initial bracket
    b ← 1                                       ▷ Initial bracket
    τ ← (1+√5)/2                                ▷ Golden ratio
    x_1 ← a + (1 − τ)(b − a)                    ▷ Initial sub-bracket
    x_2 ← a + τ(b − a)                          ▷ Initial sub-bracket
    for j ∈ {0, . . . , N − 1} do               ▷ For each atom
        X_1[j] ← X[j] + αx_1 d                  ▷ Compute coordinates at x_1
        X_2[j] ← X[j] + αx_2 d                  ▷ Compute coordinates at x_2
    end for
    f_1 ← Energy(X_1, G, C)                     ▷ Compute energy at X_1
    f_2 ← Energy(X_2, G, C)                     ▷ Compute energy at X_2
    while |b − a| > ϵ do                        ▷ While bracket is not sufficiently small
        if f_1 < f_2 then
            b ← x_2                             ▷ If f_1 < f_2 then minimum is within [a, x_2]
            x_2 ← x_1
            f_2 ← f_1                           ▷ Avoids re-evaluating Energy at X_1
            x_1 ← a + (1 − τ)(b − a)            ▷ Update sub-bracket
            for j ∈ {0, . . . , N − 1} do
                X_1[j] ← X[j] + αx_1 d          ▷ Update coordinates at new x_1
            end for
            f_1 ← Energy(X_1, G, C)             ▷ Compute energy at X_1
        else
            a ← x_1                             ▷ If f_1 ≥ f_2 then minimum is within [x_1, b]
            x_1 ← x_2
            f_1 ← f_2                           ▷ Avoids re-evaluating Energy at X_2
            x_2 ← a + τ(b − a)                  ▷ Update sub-bracket
            for j ∈ {0, . . . , N − 1} do
                X_2[j] ← X[j] + αx_2 d          ▷ Update coordinates at new x_2
            end for
            f_2 ← Energy(X_2, G, C)             ▷ Compute energy at X_2
        end if
    end while
    return (a + b)/2                            ▷ Return optimal step length
end function
```

As with previous algorithms, the sequential GSS algorithm has three surface-level candidate for-loops to parallelise, and so they have been turned into parallel loops in Algorithm 13. The while loop governed by the convergence criterion $\|b - a\| > \epsilon$ is a non-issue for task parallelism, but since we are interested in lockstep parallelism we simply choose to do a statistically sufficient number of iterations, 20, was selected through trial and error.

Algorithm 13 Parallel Golden Section Search

```
function GoldenSectionSearch(X, G, C d)
    X_1                                      ▷ N × 3 System coordinates at x_1
    X_2                                      ▷ N × 3 System coordinates at x_2
    a ← 0                                    ▷ Initial bracket
    b ← 1                                    ▷ Initial bracket
    τ ← (1+√5)/2                             ▷ Golden ratio
    x_1 ← a + (1 − τ)(b − a)
    x_2 ← a + τ(b − a)
    for j ∈ {0, . . . , N − 1} do in lockstep
        X_1[j] ← X[j] + αx_1 d[j]            ▷ Compute coordinates at x_1
        X_2[j] ← X[j] + αx_2 d[j]            ▷ Compute coordinates at x_2
    end for
    f_1 ← Energy(X_1, G, C)                  ▷ Compute energy at X_1
    f_2 ← Energy(X_2, G, C)                  ▷ Compute energy at X_2
    for i ∈ {0, . . . 20} do                 ▷ Perform 20 iterations
        if f_1 < f_2 then                    ▷ if energy at x_1 is lower than energy at x_2
            b ← x_2                          ▷ If f_1 < f_2 then minimum is within [a, x_2]
            x_2 ← x_1
            f_2 ← f_1                        ▷ Avoids re-evaluating Energy at X_1
            x_1 ← a + (1 − τ)(b − a)         ▷ Update sub-bracket
            for j ∈ {0, . . . , N − 1} do in lockstep
                X_1[j] ← X[j] + αx_1 d[j]    ▷ Update coordinates at new x_1
            end for
            f_1 ← Energy(X_1, G, C)          ▷ Compute energy at X_1
        else
            a ← x_1                          ▷ If f_1 ≥ f_2 then minimum is within [x_1, b]
            x_1 ← x_2
            f_1 ← f_2                        ▷ Avoids re-evaluating Energy at X_2
            x_2 ← a + τ(b − a)               ▷ Update sub-bracket
            for j ∈ {0, . . . , N − 1} do in lockstep
                X_2[j] ← X[j] + αx_2 d[j]    ▷ Update coordinates at new x_2
            end for
            f_2 ← Energy(X_2, G, C)          ▷ Compute energy at X_2
        end if
    end for
    return (a + b)/2                         ▷ Return the average of the bracket
end function
```

One caveat remains to the claim that Algorithm 13 is a lockstep parallel algorithm, we must show the `Energy` function is a collective operation performed in parallel.

We now turn our attention to the core computation, the energy of the system. We present first the sequential algorithm Algorithm 14 this algorithm is a pseudocode representation of the forcefield developed in python by Pedersen et al.[12]

The sequential energy computation algorithm is shown in Algorithm 14. The algorithm

consists of two nested for loops, the outer loop iterates over each atom in the system and the inner loop iterates over each neighbour of the atom. The algorithm computes the bond, angle and dihedral energies for each atom and sums them to the total energy. $\boldsymbol{X}$ is the position of each atom in the system, $\boldsymbol{G}$ denotes the adjacency list, $\boldsymbol{R}_0$ is the equilibrium bond length, $\boldsymbol{\theta}_0$ is the equilibrium angle, ϕ_0 is the equilibrium dihedral angle, $\boldsymbol{K}_{bond}$ is the bond force constants, $\boldsymbol{K}_{ang}$ is the angle force constants and $\boldsymbol{K}_{dih}$ is the dihedral force constants. Passing in all these arguments makes the function signature quite bloated, and we can simplify it by passing in the forcefield parameters in two structs, one for the force constants and equilibrium values and one for the adjacency list. For pseudocode, however, we have kept the function signature explicit.

Algorithm 14 Sequential Energy Computation

1: **function** ENERGY($\boldsymbol{X}, \boldsymbol{G}, \boldsymbol{C}$)
2: $E_{total} = 0$ ▷ Total energy of the system
3: **for** $a \in \{0, \ldots, N-1\}$ **do** ▷ For each atom
4: **for** $k \in \{0, 1, 2\}$ **do** ▷ For each neighbour
5: $b \leftarrow \boldsymbol{G}[a, k]$ ▷ kth neighbour of atom a
6: $c \leftarrow \boldsymbol{G}[a, (k+1)\%3]$ ▷ $(k+1)^{th}$ neighbour of atom a
7: $d \leftarrow \boldsymbol{G}[a, (k+2)\%3]$ ▷ $(k+2)^{th}$ neighbour of atom a
8: $\boldsymbol{ab} \leftarrow \boldsymbol{X}[b] - \boldsymbol{X}[a]$ ▷ Unit vector from atom a to atom b
9: $\boldsymbol{ac} \leftarrow \boldsymbol{X}[c] - \boldsymbol{X}[a]$ ▷ Unit vector from atom a to atom c
10: $\boldsymbol{bc} \leftarrow \boldsymbol{X}[c] - \boldsymbol{X}[b]$ ▷ Unit vector from atom b to atom c
11: $\boldsymbol{cd} \leftarrow \boldsymbol{X}[d] - \boldsymbol{X}[c]$ ▷ Unit vector from atom c to atom d
12: $cos(\beta) \leftarrow \frac{\widehat{\boldsymbol{ab}} \times \widehat{\boldsymbol{cb}}}{\sqrt{1 - \widehat{\boldsymbol{ab}} \cdot \widehat{\boldsymbol{cb}}}} \cdot \frac{\widehat{\boldsymbol{db}} \times \widehat{\boldsymbol{cb}}}{\sqrt{1 - \widehat{\boldsymbol{db}} \cdot \widehat{\boldsymbol{cb}}}}$ ▷ Dihedral angle between atoms a, b, c, d
13: $r_{ab} \leftarrow \|\boldsymbol{ab}\|$ ▷ Distance between atoms a and b
14: $E_{bond} \leftarrow \frac{1}{2}\boldsymbol{K}_{bond}[a, k](r_{ab} - \boldsymbol{R_0}[a, k])^2$ ▷ Bond energy
15: $E_{angle} \leftarrow \frac{1}{2}\boldsymbol{K}_{ang}[a, k](\widehat{\boldsymbol{ab}} \cdot \widehat{\boldsymbol{ac}} - \boldsymbol{\theta_0}[a, k])^2$ ▷ Angle energy
16: $E_{dih} \leftarrow \frac{1}{2}\boldsymbol{K}_{dih}[a, k](cos(\beta) - \boldsymbol{\Phi_0}[a, k])^2$ ▷ Dihedral energy
17: $E_{total} \leftarrow E_{total} + E_{bond} + E_{ang} + E_{dih}$ ▷ Add energy contributions to total energy
18: **end for**
19: **end for**
20: **return** E_{total} ▷ Return total energy
21: **end function**

The only computation which is dependent on the previous iteration is the summation of the energy contributions to the total energy. This summation can be performed in parallel using the reduction operation. As such we present a parallel energy computation algorithm in Algorithm 15.

Algorithm 15 Parallel Energy Computation

```
1: function ENERGY(X, G, C)
2:     E                                          ▷ N × 1 Energy contributions from each atom
3:     for a ∈ {0, ..., N − 1} do in lockstep                                 ▷ For each atom
4:         E[a] = 0                                  ▷ Initialise energy contribution to zero
5:         for k ∈ {0, 1, 2} do                                          ▷ For each neighbour
6:             b ← G[a, k]                                          ▷ k^th neighbour of atom a
7:             c ← G[a, (k + 1)%3]                            ▷ (k + 1)^th neighbour of atom a
8:             d ← G[a, (k + 2)%3]                            ▷ (k + 2)^th neighbour of atom a
9:             ab ← X[b] − X[a]                             ▷ Unit vector from atom a to atom b
10:            ac ← X[c] − X[a]                             ▷ Unit vector from atom a to atom c
11:            bc ← X[c] − X[b]                             ▷ Unit vector from atom b to atom c
12:            cd ← X[d] − X[c]                             ▷ Unit vector from atom c to atom d
13:            cos(β) ← (ab^ × cb^)/sqrt(1 − ab^·cb^) · (db^ × cb^)/sqrt(1 − db^·cb^)
                                                  ▷ Dihedral angle between planes n^_abc and n^_bcd
14:            E_bond ← 1/2 K_bond[a, k](‖ab‖ − R_0[a, k])^2                      ▷ Bond energy
15:            E_angle ← 1/2 K_ang[a, k](ab^ · ac^ − Θ_0[a, k])^2                ▷ Angle energy
16:            E_dih ← 1/2 K_dih[a, k](cos(β) − Φ_0[a, k])^2                  ▷ Dihedral energy
17:            E[a] ← E[a] + E_bond + E_angle + E_dih                ▷ Sum energy components
18:        end for
19:    end for
20:    return reduce(E, +)                                  ▷ Sum all energy contributions
21: end function
```

As is apparent from Algorithm 15 the computation of the energy contributions related to each atom is entirely independent of each other and thus parallelisable. Thus, replacing the summation of the energy contributions to the total energy with a reduction operation over individual atomic energy contributions $\boldsymbol{E}$ yields a parallel energy algorithm which can be called as a collective operation by all processing elements devoted to a given isomer.

Piecing together the conjugate gradient descent Algorithm 10, the gradient computation Algorithm 11, golden section search Algorithm 13 and the energy computation Algorithm 15, we obtain the complete parallel forcefield optimisation algorithm.

4.3 Pipeline 1: CUDA/C++ Forcefield

In Section 4.2 we described how one might parallelize forcefield optimisation on both isomerspace levels and the level of individual isomers, and how the algorithms can be executed in lockstep. In this section we aim to exemplify how parts of the algorithms described in Section 4.2 can be implemented in CUDA/C++, and how they differ from the original python implementation by Pedersen,[14] it is adapted from.

4.3.1 Data Structures

Much of the pseudocode shown in Section 4.2 gives the impression that everything is expressed in terms of functions calling other functions, and while this approach is good for describing a general algorithm that could be implemented in any language, in implementation it is convenient to introduce data structures which encapsulate certain parts of the algorithm. In the CUDA/C++ implementation, we have introduced the following data structures:

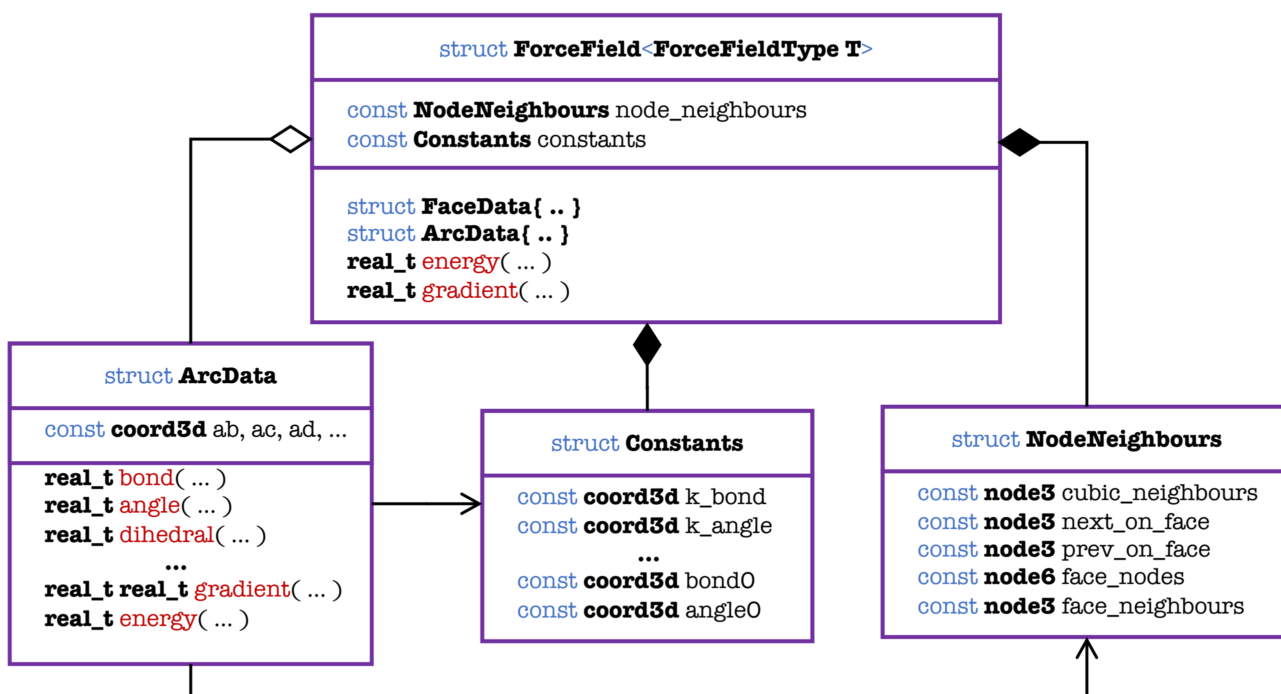


Figure 4.5: *UML diagram of the data structures used in the CUDA/C++ implementation of the forcefield optimisation. Each thread on the device has a* `ForceField` *structure which contains one (UML: Composition)* `Constants` *and* `NodeNeighbours` *structure containing all the static information which is required to compute the energy contribution from the corresponding atom, as well as the gradient component with respect to said atom.* `ArcData` *is a helper structure created by* `ForceFields` *in its* ***energy****() and* ***gradient****() methods.* `ArcData` *initialises its state through access to (UML: Association)* `NodeNeighbours` *and uses* `Constants` *in its calculations.*

`ForceField` is the overarching container for the shared state between the algorithms, the motivation is that Algorithm 10, Algorithm 13, Algorithm 15 and Algorithm 11 all access the same constants and the same graph information, and the data is untouched

once it has been initialised. In the aforementioned algorithms the adjacency information, $\boldsymbol{G}$, is shown as a shared variable that is only necessary for the initialization of the Constants (Algorithm 8). After the initialization each node actually only needs to store the indices of its neighbours, the outer neighbours $b_m, b_p, c_m \ldots$ and face neighbours if flatness is used, more on that later. So it turns out we can make do with local adjacency information, which we store in the `NodeNeighbours` struct. Similarly, the `Constants` struct contains all the constants that are used in the energy and gradient computations. It is important to note that these are not just design choices, they carry performance implications, as storing information locally avoids the need for global memory access or excessive cache use.

Now Algorithm 15 and Algorithm 11 could be implemented exactly as described in the pseudocode, but we have chosen to encapsulate the interior of the neighbour loops $k \in \{0, 1, 2\}$ in a structure `ArcData` and with each equation in Section 2.2 corresponding to a function in the `ArcData` struct. The initialization of state, lines [9, 18] in Algorithm 11 is conveniently handled by the constructor. This is done to make the code more readable and easier to extend with new equations, but it is ultimately a stylistic choice.

We have to mention that while coordinates $\boldsymbol{X}$ are also used across the algorithms, it is not stored in the `ForceField` struct, the rationale is two-fold: first, the functions depend on the coordinates of the nodes and the state is modified by the functions, secondly storing these coordinates as a mutable state in `ForceField` was found to yield worse performance, presumably because the compiler is unable to make certain optimisations.

4.3.2 Device Side Encapsulation

As mentioned in the CUDA programming model Section 3.1.2 we distinguish between code that runs on the GPU as *Device* code and the calling code running on the CPU, as the *Host* code. Now in typical CUDA/C++ code bases, there exists a lot of boilerplate setup code which is required to get the GPU to do work. This may be an acceptable consequence if one intends to write mostly self-contained scripts where host and device code exist within the same file and compilation unit, however, the intention was for this code to be integrated as an optional component in the *Fullerene* program. As such some design choices were required to make the code both easier to interact with, maintain and extend.

The most pressing concern in any *Host* - *Device* interaction is the transfer of data between the two. In CUDA/C++ this process is quite verbose and error-prone, you are required to declare *Host* pointers and device pointers, allocate both of copying between them then requires you to specify the number of bytes you want to copy and the direction of the type of copy operation you wish to perform. This becomes incredibly cumbersome especially if you have many arrays you wish to copy back and forth. This is what motivated the design of the `IsomerBatch` structure which is the central data container in which all isomers are stored and all GPU operations defined here are performed on. It stores 4 arrays of data, the coordinates $(M_B \times N \times 3)\boldsymbol{X}_{in}$, cubic graphs,

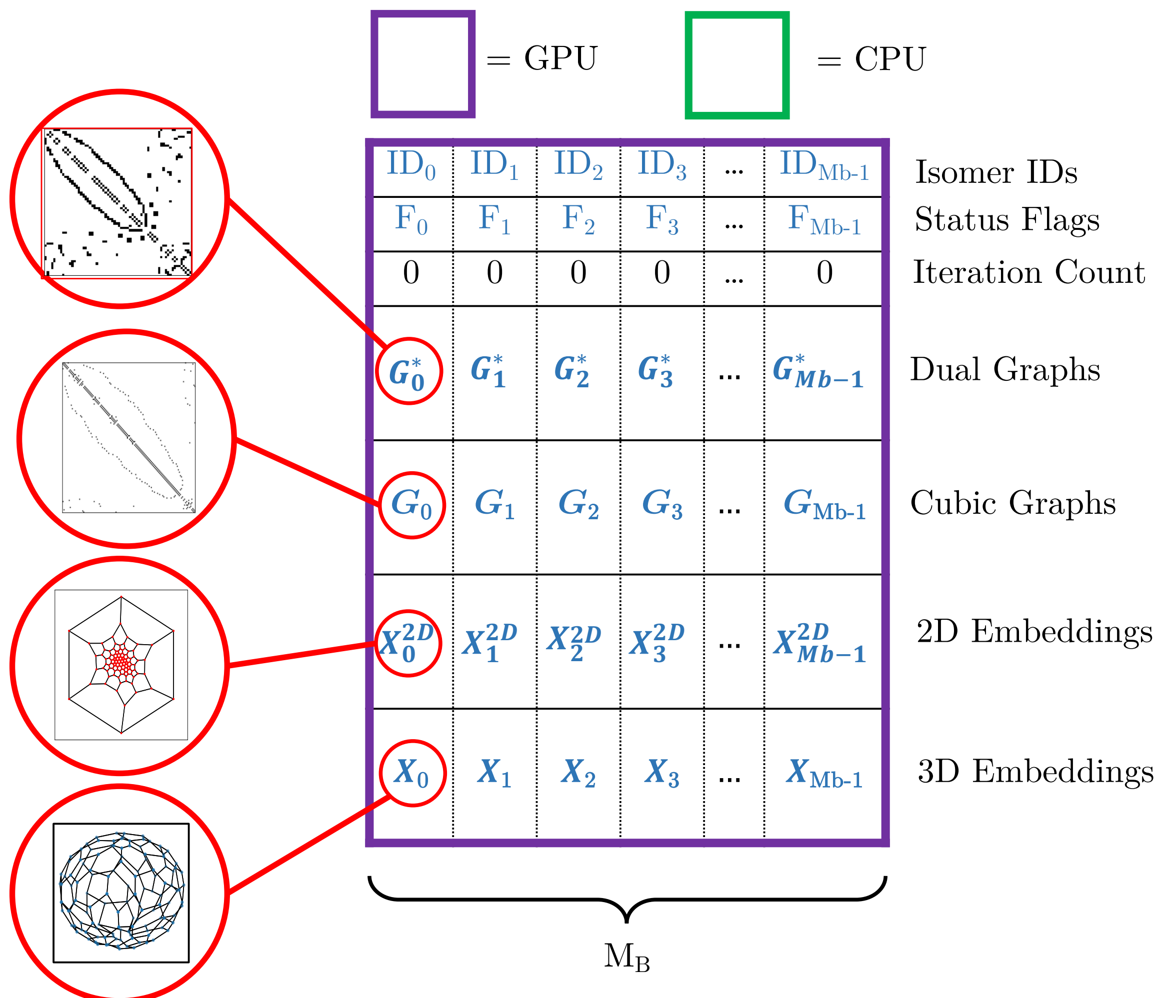


Figure 4.6: *Anatomy of the* `IsomerBatch`*, the red pictures are visualizations of the data, dual and cubic graphs are pictured as dense adjacency matrices but are of course stored in sparse form. 2D and 3D embeddings exist as coordinates, so the visual representation is accurate here. We use* ***purple*** *to signify that the batch resides in GPU memory and* ***green*** *for CPU memory.*

$(M_B \times N \times 3)\boldsymbol{G_{in}}$, dual graphs $(M_B \times N_f \times 6)\boldsymbol{G^*_{in}}$, 2D embeddings $(M_B \times N \times 2)\boldsymbol{X^{2D}_{in}}$ as well as three sets of metadata all the dimension $(M_B \times 1)$: id-list $\boldsymbol{ID}$, status flags $\boldsymbol{F}$ and iteration counter $\boldsymbol{It}$. The anatomy of the `IsomerBatch` is shown in Fig. 4.6. We will use this visualization of the IsomerBatches in the following sections to illustrate the data flow.

We define a set of natural utility methods that act on `IsomerBatch` structures, copy, resize and sort. The function signatures are shown in Fig. 4.7.

The user can specify the stream and corresponding GPU which operations should be enqueued on, through the `LaunchCtx` struct. Furthermore, whether the operation should be blocking or non-blocking can be specified through the `LaunchPolicy` enumerable. The default behaviour, if nothing is specified, is to execute the operation on the default stream on the default GPU and to block until previous enqueued operations have com-

```
void copy (IsomerBatch&, const IsomerBatch&, const LaunchCtx&, const LaunchPolicy, const std::pair<int,int>, const std::pair<int,int>)
void resize (IsomerBatch&, const LaunchCtx&, const size_t, const LaunchPolicy)
void sort (IsomerBatch& B, const BatchMember, const SortOrder)
```

Figure 4.7: *Function signatures of some primary utility functions on the* `IsomerBatch` *struct.*

pleted. The implementation of `LaunchCtx` can be found in `launch_ctx.cu`. We shall see in Section 5.4 how asynchronous execution enables pipeline parallelism and improves performance.

The culmination of all this is that we can now express isomerspace optimisation very simply as an extension to the *Fullerene* program. The code is shown in Listing 4.1

```
FullereneDual G(Nf);//Graph for BuckyGen to fill
IsomerBatch B0(N,M_b,BufferType::HOST_BUFFER);   //Host batch
IsomerBatch B1(N,M_b,BufferType::DEVICE_BUFFER);//Device batch
while (more_to_do)
{
    while (B0.size() ¡ B0.capacity())
    {
        //Generate next fullerene
        more_to_do &= BuckyGen::next_fullerene(Q,G);
        if (!more_to_do) break;
        G.update();
        PlanarGraph pG = G.dual_graph();     //Compute Cubic Graph
        pG.layout2d = pG.tutte_layout();     //Compute 2D Embedding
        Polyhedron P(pG);                    //Create polyhedron
        P.points = P.zero_order_geometry(); //Compute 3D Embedding
        B0.append(P, I);                     //Append to batch
        I++;
    }
    if (B0.size() == 0) break;
    cuda_io::copy(B1, B0); //Copy to device B1 ¡− B0
    //Forcefield optimisation
    isomerspace_forcefield::optimize¡PEDERSEN>(B1, N*5, N*5);
    B0.clear();
    //Do something with results from B1 next... (Future work)
}
```

Listing 4.1: *Script for Pipeline 1: Lockstep Forcefield optimisation. Fullerenes are produced by* `BuckyGen`*,then dualized, embedded and projected using methods in the Fullerene program. We then append the* `Polyhedron` *to an IsomerBatch and finally copy the batch to a device batch and call optimize on this batch.*

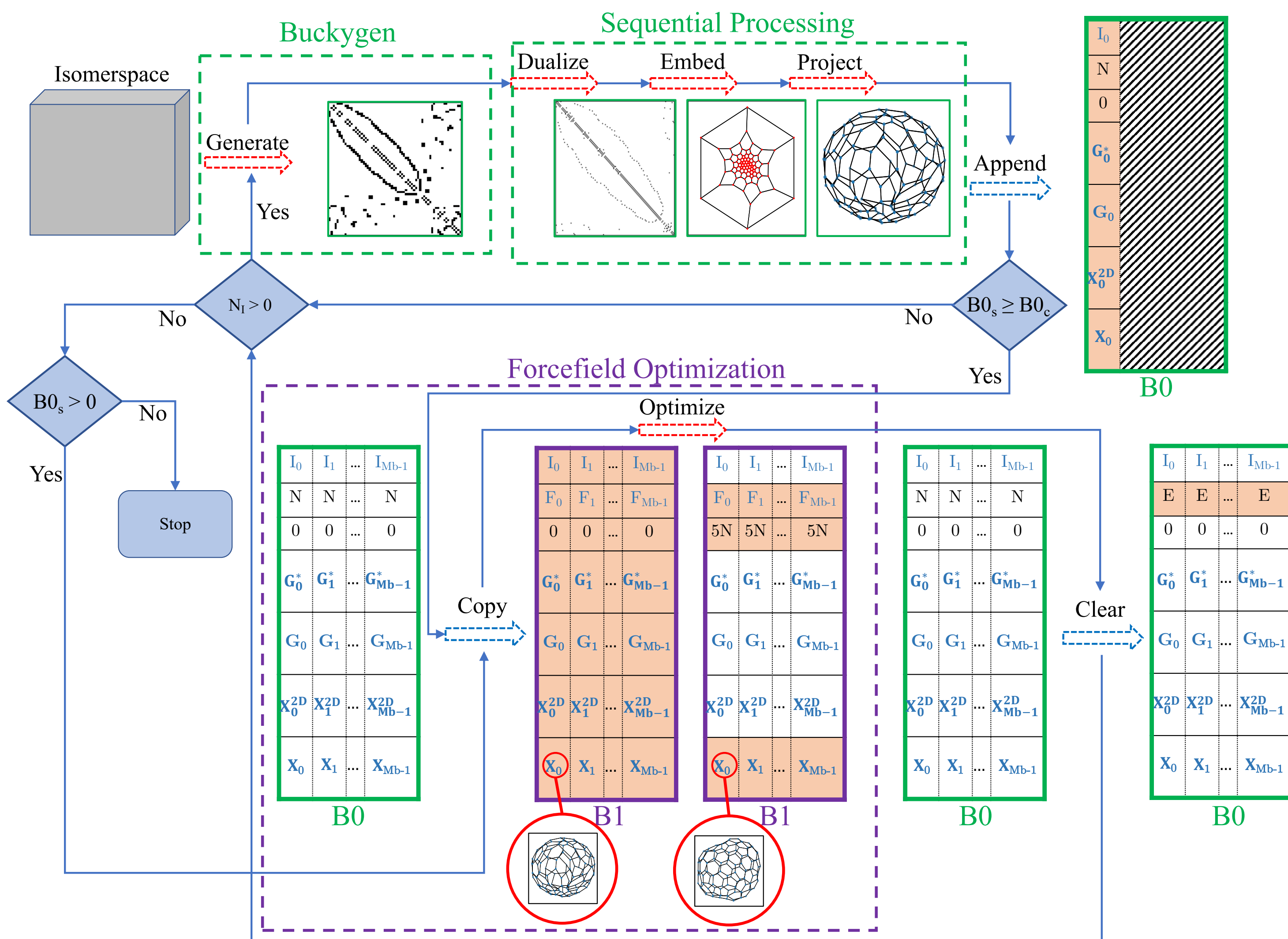


Figure 4.8: *Pipeline 1: Parallel Lockstep forcefield optimisation, everything else is sequential. The figure shows a snapshot of the state of the* `IsomerBatch` *structures throughout the first iteration of Listing 4.1. Orange coloured fields indicate that they were modified by the previous operation, grey implies that the memory is default initialised.*

Fig. 4.8 is a visual representation of the first pipeline (Listing 4.1) that is part flow-chart part state-diagram, we use the same visual representation of `IsomerBatch` as in Fig. 4.6 to illustrate the state of the `IsomerBatch` structures throughout the first iteration of the pipeline. We use **green** to imply that data resides on the CPU and areas enclosed in **green** dashed lines indicate that operations within it are carried out on the CPU. Similarly, **purple** is used to show that data resides on the GPU and that operations are carried out on the GPU.

With the implementation of the parallel forcefield optimisation complete we must assess correctness and convergence.

4.4 Validation and Convergence

In the preceding sections we have shown how the forcefield optimisation algorithm can be implemented conceptually on any massively parallel hardware, (Algorithm 10, Algorithm 13, Algorithm 11) as well as how we can concretely implement it in CUDA/C++ and incorporate it in the *Fullerene* program (Listing 4.1). We have yet to show that the implementation is valid and that the results are comparable to the results of the sequential implementation.

4.4.1 Validation

To do this we selected random $\mathsf{min}(number_of_isomers(C_N), 1000)$ isomers from each isomerspace $C_{20}, C_{24}, \cdots, C_{200}$ and performed forcefield optimisation on them using both the sequential Fortran implementation by Wirz et al. and the new parallel forcefield optimisation implementation. Ideally we would be able to compare coordinates directly and measure their variation. However, Wirz et al. used a combination of conjugated gradient descent and brent's method, and we used golden section search with fixed iteration count. Therefore, comparing coordinates directly would not yield anything meaningful. Instead, we compare the relative root mean squared error (RRMSE) of the bond lengths, angles and dihedrals w.r.t equilibrium values. We perform this analysis using both the gradient corrected forcefield by Pedersen[12] and the original forcefield by Wirz et al.[18]

The results of this analysis are represented in Fig. 4.9 as sets of histograms. We see that the shape of the histograms are very similar for all three forcefields, notably the original Wirz forcefield implementation gives rise to a smear of outlier isomers with higher RRMSE values. These appear to be absent in the other two forcefields. A caveat to this is that the CUDA/C++ implementation produces a greater volume of invalid geometries full of NaNs which cannot be shown in a histogram. Perhaps counterintuitively, this analysis reveals that RRMSE values of bond lengths and angles decrease as $C_N \rightarrow C_{200}$ indicating that the forcefield is converging to a more accurate representation of the true potential energy surface. We presume that two factors are responsible for this, firstly the total Gaussian curvature of the fullerenes can be distributed over a greater number of faces and secondly the fraction of fullerenes that follow the *isolated pentagon rule* (IPR), which is known to improve thermodynamic stability,[9] increases as $C_N \rightarrow C_{200}$.

In order to be able to more directly compare the results we summarise the information in Fig. 4.9. The distributions are not normal distributions, therefore we plot the median and the 25th and 75th percentiles of the RRMSE values for each isomerspace C_N in Fig. 4.10, instead of means and standard deviations. From Fig. 4.10 we see that the CUDA/C++ implementation of the Wirz forcefield is comparable to the sequential implementation. Curiously, Wirz + GSS seems to find minima with lower Angle RRMSE but slightly higher Dihedral RRMSE values. We do not know why this occurs, this might be worth investigating further. In any case, the CUDA/C++ implementation of the Pedersen forcefield seems to converge closer to the equilibrium parameters than the Wirz forcefield.

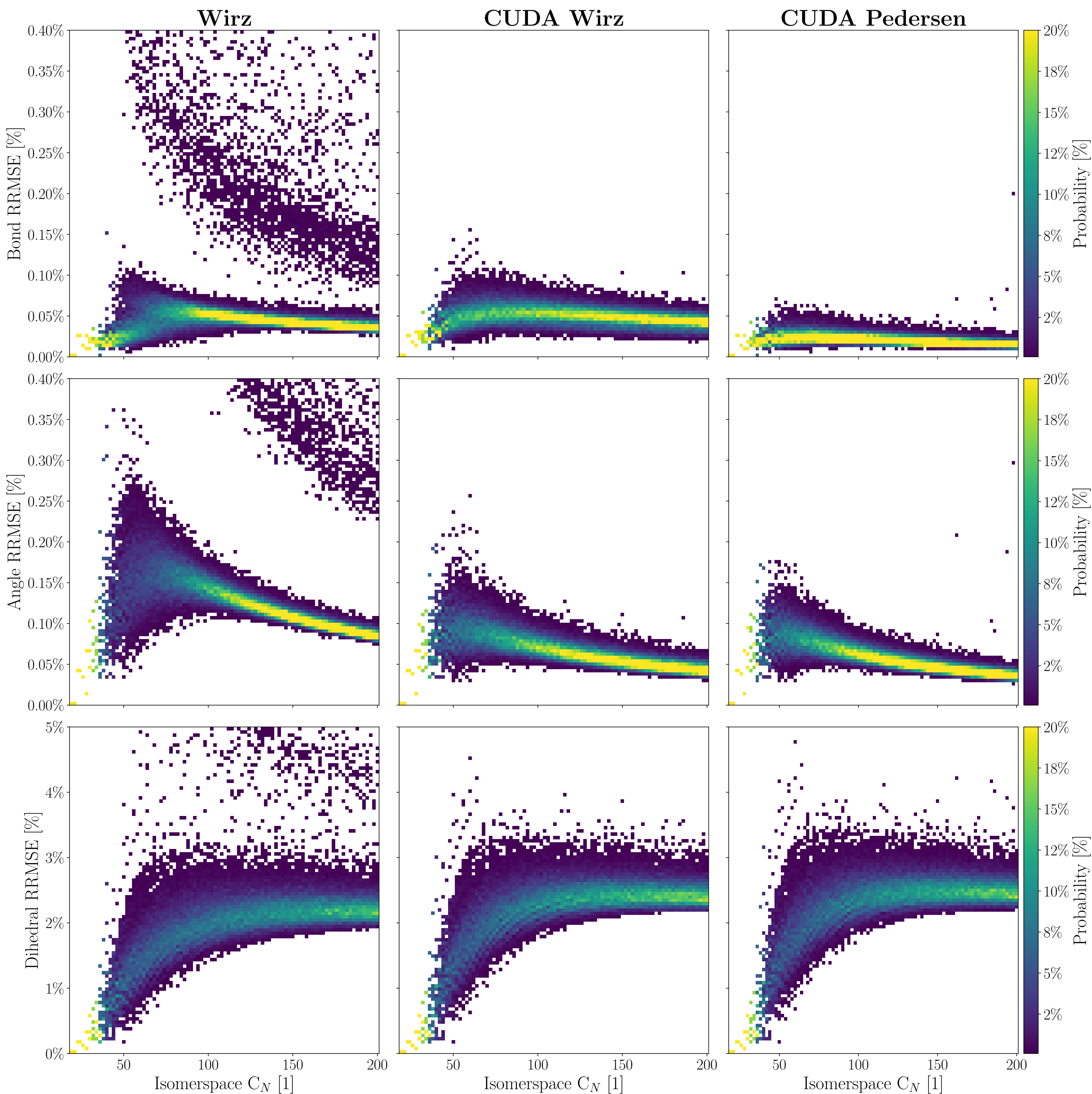


Figure 4.9: *RRMSE of bond lengths, angles and dihedrals for Original Wirz forcefield as well as CUDA/C++ implementations of Wirz + GSS and Pedersen + GSS. Each plot contains a histogram of RRMSE values for each isomerspace C_N, the probability is the fraction of isomers in an isomerspace that fall within a given RRMSE bin. Bins were linearly distributed from the minimum to the maximum value of the y-axes in each plot.*

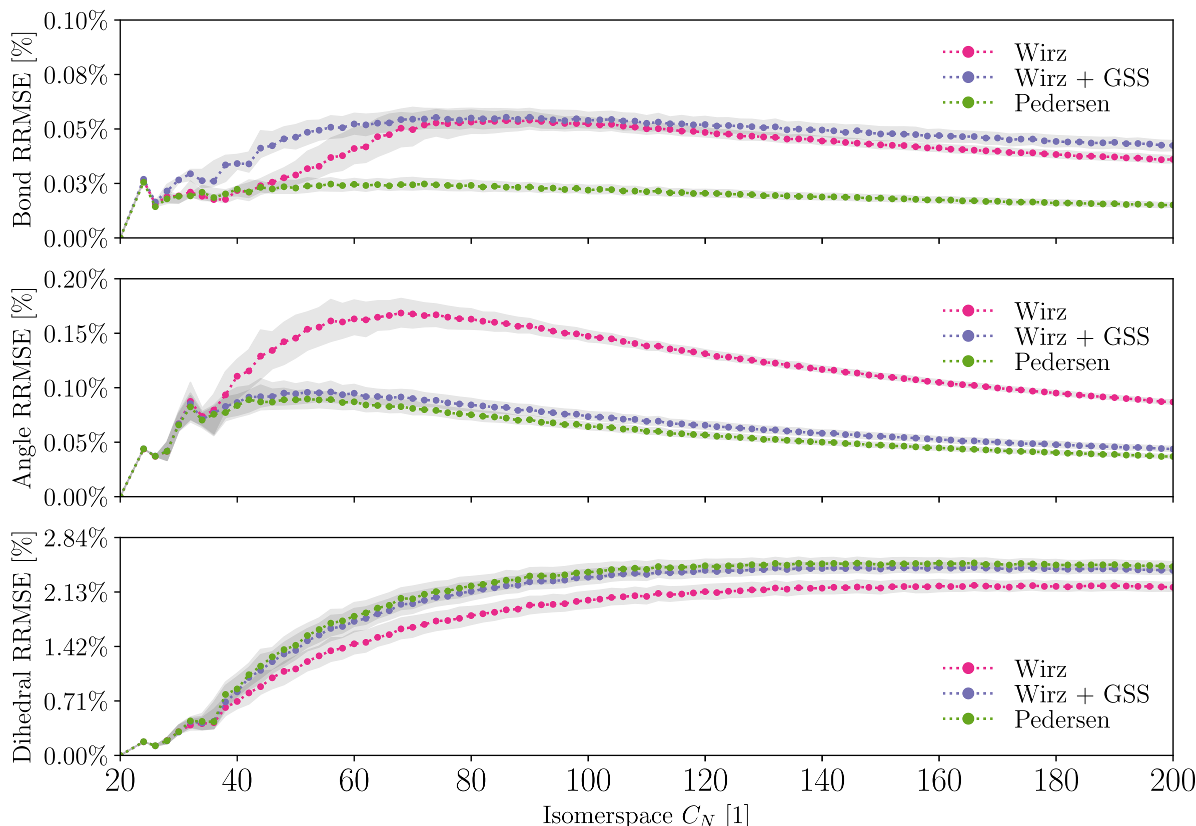


Figure 4.10: *Medians of RRMSE of bond lengths, angles and dihedrals for Original Wirz forcefield as well as CUDA/C++ implementations of Wirz + GSS and Pedersen + GSS. Shaded areas are encompassed by the 25th and 75th percentiles of the RRMSE values for each isomerspace C_N.*

4.4.2 Convergence

So far we have discussed only how the forcefield optimisation itself may be performed in parallel on massively parallel hardware, we now turn out attention to the convergence of the optimisation algorithm itself.

We devised a script for optimizing isomers from the randomly sampled data, we perform optimisations in steps of $1/10N$ iterations until an upper limit of $50N$ iterations is reached. After each optimisation step we push any finished isomers to an output queue (the implementation of this will be discussed later) this way we are able to keep track of the number of iterations required to converge each isomer. Fig. 4.11 shows the resulting probability distribution of the number of iterations required to converge an isomer as a function of isomerspace C_N.

While it does appear that the number of iterations required does increase super-linearly the incline is slight and appears to drop off, as if to suggest something of the order $Nlog(N)$ iterations required. Furthermore, we notice that a small spread of the distribution can be observed for all isomerspaces, specifically the difference between the 50th percentile and the 99th percentile is around $1 \times N$ iterations for most isomerspaces.

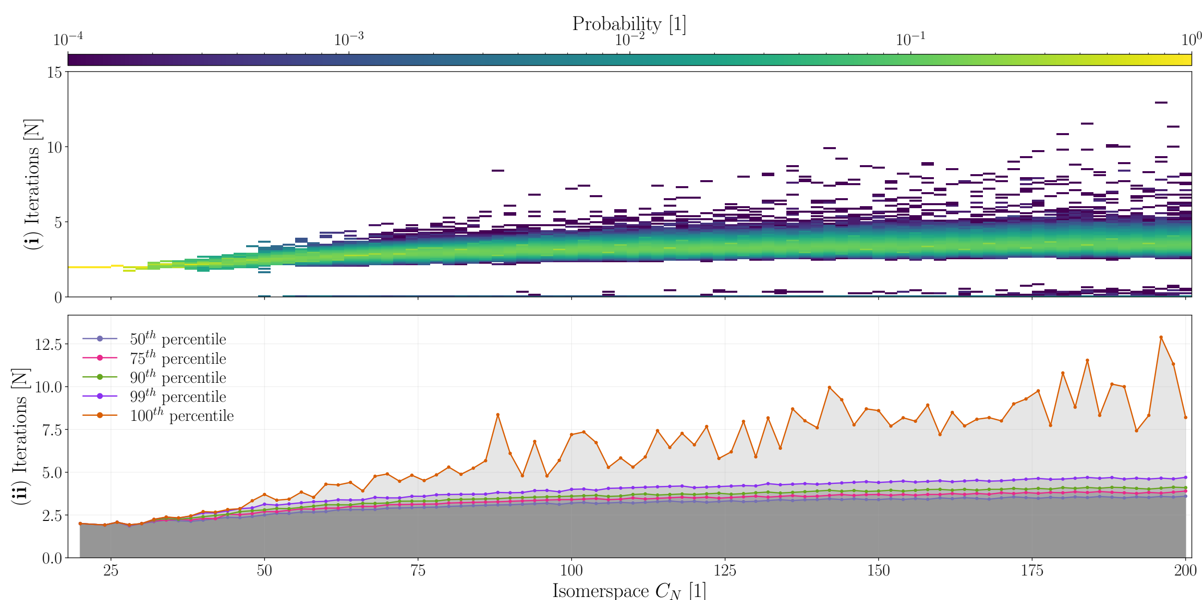


Figure 4.11: *(**i**) Logarithmic probability map showing the probability that an isomer requires a given amount of iterations, the y-axis shows iteration count in multiples of N. (**ii**) shows the required iterations to converge a fraction of the isomers, 50th (blue), 75th (pink), 90th (green), 99th (purple) and 100th (orange) percentiles respectively.*

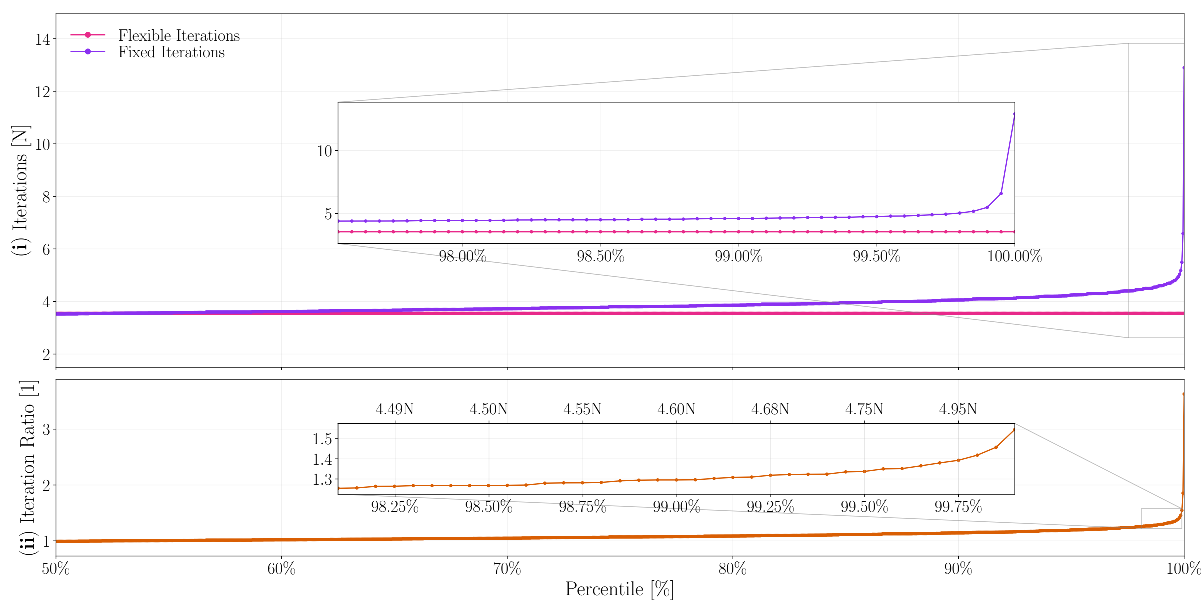


Figure 4.12: *(**i**) Number of iterations required to converge given either fixed number of iterations (Purple) or variable number of iterations (Pink) as a function of the percentile of isomers we wish to optimise. These statistics are based on the 200k randomly sampled isomers in the isomerspace range $C_{160}, \cdots, C_{200}$. (**ii**) The ratio (orange) between the fixed number of iterations required and average number of iterations required, given flexibility, is shown as a function of percentile.*

We now look at the data in a slightly different way by plotting the number of fixed iterations we would need to perform as a function of the percentile of isomers that we wish to successfully optimise, as well as the average number of iterations required to optimise a given percentile of isomers, given we are able to facilitate variable number of iterations for each isomer.

We see that the per isomer, the ratio between, performing a fixed number of iterations across all isomers and performing a variable number of iterations for each isomer, grows as a function of the percentile of isomers we wish to have converged. The ratio grows very slowly initially but naturally picks up as we approach the 100th percentile. The work by Pedersen[12] established a heuristic of $3N$ on the basis of a curated set of high symmetry isomers, but we see here that $5N$ is really more appropriate to account for the vast majority of isomers. If we set $5N$ as the number of iterations to perform in a fixed-iteration isomerspace optimisation scheme, we would have in theory 40% performance to gain from a variable-iteration scheme. This is the primary motivation for designing a parallel queue, which we shall see in the next section.

4.5 Pipeline 2: IsomerQueue Forcefield

As we have seen in the previous chapter, if we were able to somehow replace converged isomers from a batch with new geometries every $\mathcal{O}(N)$ steps, we would be able to theoretically achieve upwards of 40% better performance than otherwise. Additionally, it gives a flexibility in choices of optimisation algorithms as we would be able to pick an algorithm with better average case complexity even if the upper bound complexity exceeds the current algorithm. Moreover, the flexibility would let us be greedier with the choice of convergence criteria and maximum number of steps without sacrificing heavily on performance. This is the motivation that drove us to design a queue system that would allow us to replace converged isomers with new geometries every $\mathcal{O}(N)$ steps.

4.5.1 Isomer Queue

The queue system is designed to be a FIFO queue which is accessed by many GPU threads simultaneously through a set of collective operations. The queue is implemented as a circular queue, meaning that rather than constantly resizing the underlying array, we simply wrap around the array when we reach the end. This is done by keeping track of the front and back of the queue, and incrementing/decrementing them respectively. This is the typical way in which a queue is implemented.

The `IsomerQueue` defines a set of operations which can be performed on it: `refill_batch` which refills a target batch with isomers from the queue, `push` which for a given batch pushes all the converged and failed isomers onto the queue, overloaded version of `push` which inserts a new isomer or entire batch into the queue. The nomenclature is slightly different from that of usual FIFO implementations, as their purpose is slightly different.

4.5.2 Refill Batch

The `refill_batch` operation is most similar to what might conventionally be called `pop` or `dequeue` operation, but rather than returning a single element from the front of the queue, it takes the first N elements from the queue and inserts them into the first N empty slots in a target batch of isomers. To see how this works consider Algorithm 16:

The refill operation works by first checking each slot in the target batch $\boldsymbol{B}$ to see if it needs to be replaced, this can be done in parallel as each element is independent of each other. We assign a binary array $\boldsymbol{A_Q}$ to indicate whether each element needs to be replaced. We then compute the exclusive scan (see Section 3.4.2) of a copy of this array $\boldsymbol{S}$, which gives us the number of elements which need to be replaced up to the ith element. The last element of $\boldsymbol{S}$ gives us the total number of elements which need to be replaced N_r. Now we go through the $\boldsymbol{A_Q}$ in parallel and check whether each element needs to be replaced and whether its corresponding read index $\boldsymbol{S}[i]$ is within the queue. If both of these conditions are true, we replace the ith element of $\boldsymbol{B}$ with the $\mathtt{mod}(\boldsymbol{f}+\boldsymbol{S}[i],\boldsymbol{c})^{th}$ element of the queue $\boldsymbol{B_Q}$. When all this is done we need to update the queue counters $\boldsymbol{f}$, $\boldsymbol{b}$ and $\boldsymbol{s}$ to reflect the number of elements which have been removed

from the queue. The source code for this operation is shown in Listing B.6.

Algorithm 16 Refill Batch

1: **function** REFILL BATCH($\boldsymbol{B}, \boldsymbol{F}, \boldsymbol{B_Q}, \boldsymbol{f}, \boldsymbol{b}, \boldsymbol{s}, \boldsymbol{c}$) ▷ Batch, Flags, Queue Batch, front, back, size, capacity
2: $\boldsymbol{S}$ ▷ $\text{size}(B) \times 1$ Scan array for computing indices of empty slots in batch
3: $\boldsymbol{A_Q}$ ▷ $\text{size}(B) \times 1$ Binary array indicating whether each element must be replaced.
4: **for** $i \in \{0, \ldots, \text{size}(\boldsymbol{B}) - 1\}$ **do in lockstep** ▷ For loop across the queue
5: $\boldsymbol{A_Q}[i] \leftarrow \boldsymbol{F}[i]! = NOT_CONVERGED$
6: $\boldsymbol{S}[i] \leftarrow \boldsymbol{A_Q}[i]$ ▷ Store the ith value of $\boldsymbol{A_Q}$ in the $\boldsymbol{S}$
7: **end for**
8: $\text{ex_scan}(\boldsymbol{S})$ ▷ Compute the exclusive scan of $\boldsymbol{S}$
9: $N_r \leftarrow \boldsymbol{S}[\text{size}(\boldsymbol{B}) - 1]$ ▷ Last element of $\boldsymbol{S}$ indicates number of requests
10: **for** $i \in \{0, \ldots, \text{size}(B)\}$ **do in lockstep**
11: **if** $\boldsymbol{A_Q}[i] = 1$ **and** $\boldsymbol{S}[i] < \boldsymbol{s}$ **then** ▷ Check that the element is within the queue
12: $\boldsymbol{B}[i] \leftarrow \boldsymbol{B_Q}[\text{mod}(\boldsymbol{f} + \boldsymbol{S}[i], \boldsymbol{c})]$ ▷ Replace the ith element of $\boldsymbol{B}$
13: **end if**
14: **end for**
15: $e \leftarrow \boldsymbol{s} > N_r$ ▷ Check if queue has more elements than number of requests
16: **if** e **then**
17: $\boldsymbol{s} \leftarrow \boldsymbol{s} - N_r$ ▷ Subtract requests from queue size
18: $\boldsymbol{f} \leftarrow \text{mod}(\boldsymbol{f} + N_r, \boldsymbol{c})$ ▷ Increment front by number of requests
19: **else**
20: $\boldsymbol{s} \leftarrow 0$ ▷ Set queue size to 0
21: $\boldsymbol{f} \leftarrow -1$ ▷ Set front to -1
22: $\boldsymbol{b} \leftarrow -1$ ▷ Set back to -1
23: **end if**
24: **end function**

4.5.3 Push (Drain Batch)

The `push` operation is similar to the refill operation, but instead of replacing elements in a target batch, it pushes all the converged and failed isomers from a source batch onto the queue. To see how this differs from the refill operation consider the following pseudocode:

Algorithm 17 Push

```
1: function PUSH(B, F, B_Q, F_Q, f, b, s, c)  ▷ Batch, Flags, Queue Batch, Queue Flags, front, back, size, capacity
2:     S              ▷ size(B) × 1 Scan array for computing indices of empty slots in batch
3:     A_Q            ▷ size(B) × 1 Binary array indicating whether each element must be replaced.
4:     for i ∈ {0, ..., size(B) − 1} do in lockstep     ▷ For loop across the queue
5:         A_Q[i] ← F[i] = FAILED or CONVERGED
6:         S[i] ← A_Q[i]                  ▷ Store the i^th value of A_Q in the S
7:     end for
8:     ex_scan(S)                         ▷ Compute the exclusive scan of S
9:     N_r ← S[size(B) − 1]     ▷ Last element of S indicates number of requests
10:    for i ∈ {0, ..., size(B)} do in lockstep
11:        if A_Q[i] = 1 then
12:            B_Q[mod(b + 1 + S[i], c)] ← B[i]     ▷ Insert at the back of the queue
13:            F_Q[mod(b + 1 + S[i], c)] ← F[i]     ▷ Producer consumer pattern
14:            F[i] ← EMPTY
15:        end if
16:    end for
17:    if s = 0 and N_r > 0 then
18:        f ← 0                                  ▷ Set front to 0
19:        b ← N_r − 1                ▷ Set back to number of requests - 1
20:    else
21:        b ← mod(b + N_r, c)        ▷ Increment back by number of requests
22:    end if
23:    s ← s + N_r              ▷ Increment queue size by number of requests
24: end function
```

The push shares the same structure as the refill operation, but there are some key differences, firstly we do not need to check whether the read index is within the queue as we are pushing onto the queue. The check for whether an element is to be pushed is different as well as we only push elements which have converged or failed, we might want to generalise this in the future such that any condition can be specified. The index at which we insert is of course different as well, we insert at the back of the queue. The updates to the queue counters are also different as we need to increment the back counter by the number of requests N_r rather than the front, the size is simply updated by adding N_r to it. One might wonder if this operation breaks if the capacity of the queue is less than $N_r + s$ and the answer is yes, but we account for this by increasing the capacity of the queue to accommodate at most size(B) new elements, before pushing. The source code for this operation can be found in Listing B.7.

4.5.4 Host Side: Push and Pop

The `push` and `pop` are both host side operations that are used to insert singular elements from a queue or remove singular elements `Graph`, `PlanarGraph` or `Polyhedron` objects. The `pop` operation is not intended to be used in performance sensitive code, it exists more as a convenience method for testing or debugging, as it requires a host side copy of the queue to be updated and a data layout conversion to happen such that a `Polyhedron` object can be returned for further analysis in the *fullerene* program. The host side overload of the `push` operation is necessary however as we need to facilitate the initial data layout conversion of the individual `Graph` objects constructed in the *fullerene* program to be inserted in an `IsomerBatch` for parallel processing. The source code for these functions can be found in Listing B.4 and Listing B.5 respectively.

4.5.5 Host Side: Resize

The `resize` operation is a host side operation which is used to resize the queue. It is used to increase the capacity of the queue to accommodate more elements. The process of resizing any container necessarily involves allocation of new memory, copying of the old data to the new memory and deallocation of the old memory. This is generally costly, so we want to avoid it as much as possible. Resizing a circular queue container is somewhat less trivial than resizing for instance a vector container, as we need to account for the scenario where the back counter is less than the front counter. In this case we need to copy the elements in the interval $[front, capacity - 1]$ of the underlying container to the front of the new container $[0, capacity - front - 1]$, similarly the $[0, back]$ interval of the original queue must be copied to $[capacity - front, capacity - front + back]$ of the new queue. The source code implementation of this operation can be found in Listing B.3.

4.5.6 The Queue Enabled Design

We have discussed the conceptual data structure of a circular FIFO queue and gone through the fundamental collective and sequential operations which can be performed on the queue. We now wish to use this queue in conjunction with the *fullerene* program and the lockstep parallel forcefield optimisation previously discussed. Fig. 4.13 shows the queue enabled design of isomerspace optimisation.

The queue enabled design of isomerspace optimisation uses a mix of sequential and parallel methods. *BuckyGen* generates isomers which are then dualized, embedded and projected onto a sphere, using the pre-existing sequential *fullerene* program functions. These initial geometries are then pushed onto a queue **Q0**. We then proceed to refill a batch **B0** using `refill_batch`, forcefield optimizing this batch $0.5N$ steps and finally pushing any finished isomers `Push` onto a queue **Q1**. These final three steps are repeated until the queue **Q0** has fewer isomers than the size of the batch `size`(**B0**). When not enough isomers are present in **Q0** to start over producing more isomers and generating their initial geometries.

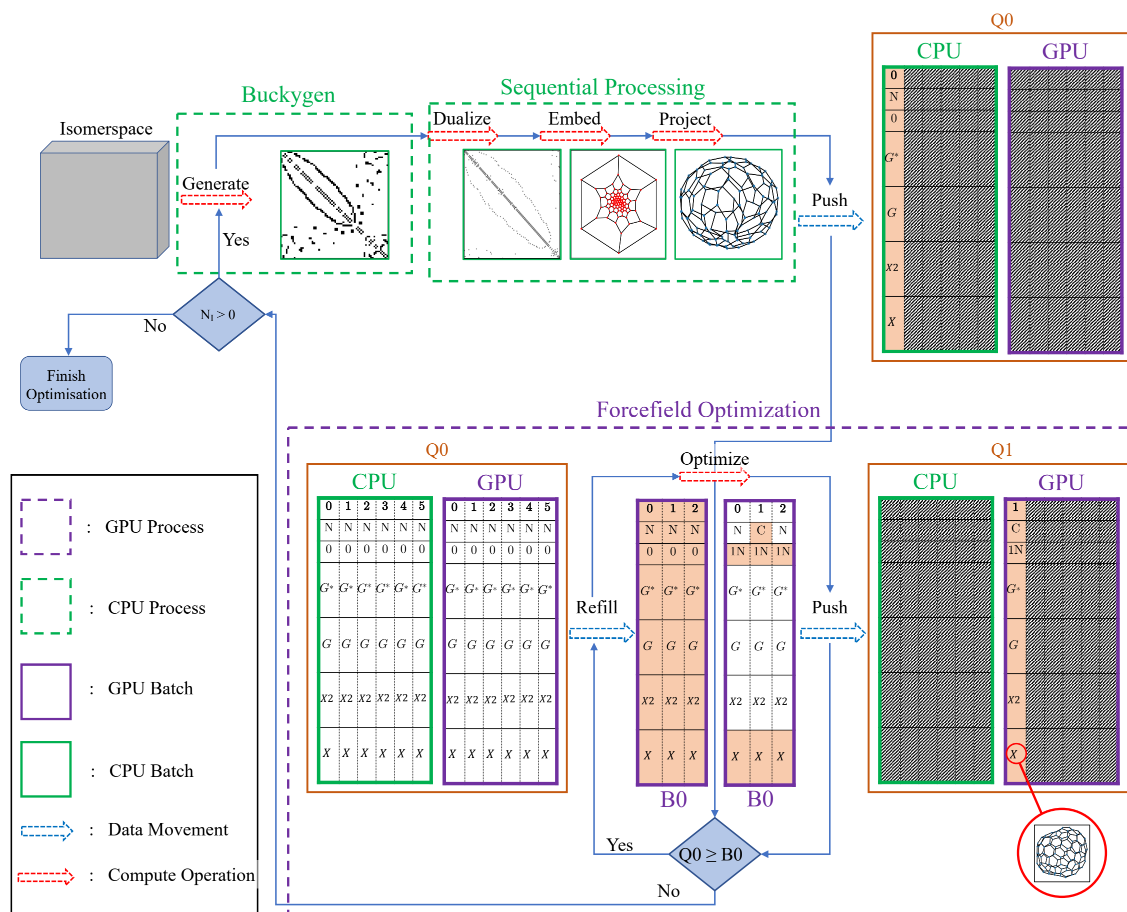


Figure 4.13: *The queue enabled design of isomerspace optimisation. The* **green** *dashed lines contain components which are sequential and pre-existing fullerene program functions. Operations contained within the* **purple** *domain are parallel operations which are executed in lockstep. The* **blue** *lines indicate the flow of control throughout the program. The dashed arrows indicate either a data transfer or a function call.*

4.6 Performance

In Section 4.4.2 we showed the variance in number of forcefield optimisation steps required for each isomer, and with it motivated the development of a parallel isomer queue to enable a faster fullerene optimisation pipeline in theory. In this section we will benchmark the performance of the forcefield optimisation section of the pipeline and compare it with the queue enabled design presented in the previous section.

4.6.1 *Forcefield Optimisation Performance*

In the introduction of this chapter we showed how the forcefield optimisation kernel is the bottleneck of the fullerene optimisation pipeline, by an order of magnitude. Let us compare the performance of the sequential forcefield optimisation kernel as implemented in Fortran and Python with the parallel forcefield optimisation kernel as implemented in CUDA.

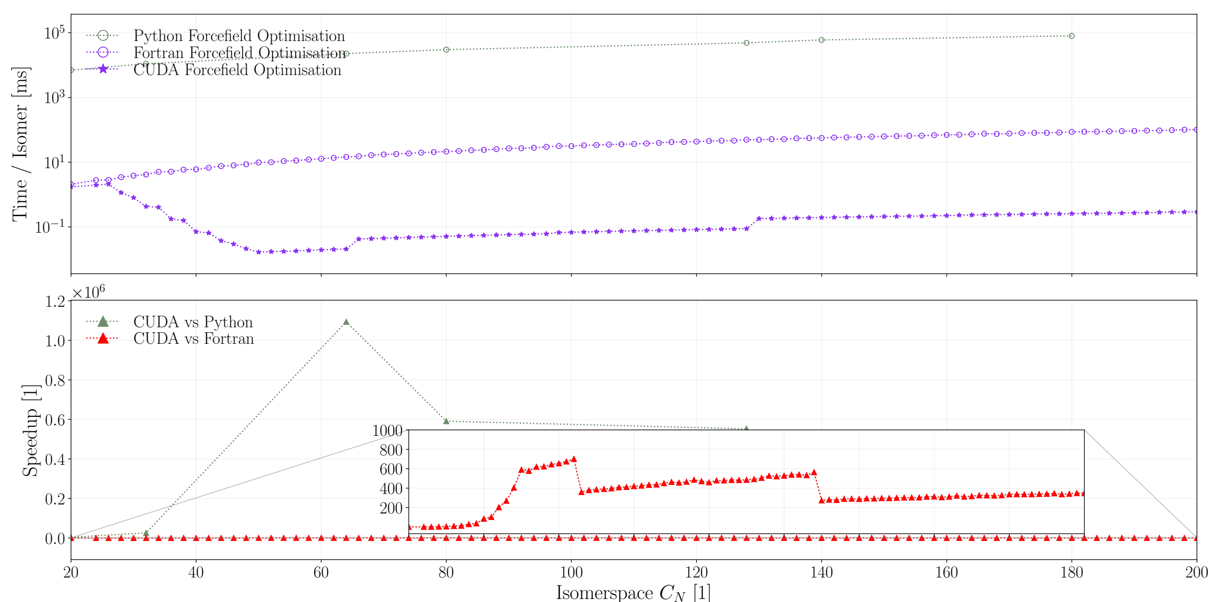


Figure 4.14: *Forcefield optimisation time per isomer as a function of isomerspace C_N for the sequential Fortran implementation, the sequential Python implementation and the parallel CUDA implementation. The speedup is shown in the subplot below.*

As we can clearly see from Fig. 4.14, the parallel CUDA implementation of the forcefield optimisation kernel is significantly faster than the sequential Fortran and Python implementations. The python implementation is somewhat of a ridiculous comparison here, showing 6 orders of magnitude speedup, as the python implementation was never built to perform. The python implementation is however mathematically the most similar to the CUDA implementation. Importantly the CUDA implementation is around $200\text{-}600\times$ faster than the pre-existing Fortran implementation, the kind of speedup that we were hoping to extract from efficient lockstep parallelisation.

4.6.2 Queued Forcefield Optimisation Performance

Now we are ready to compare the performance of the CUDA forcefield optimisation kernel with the queue enabled design presented in Section 4.5.6.

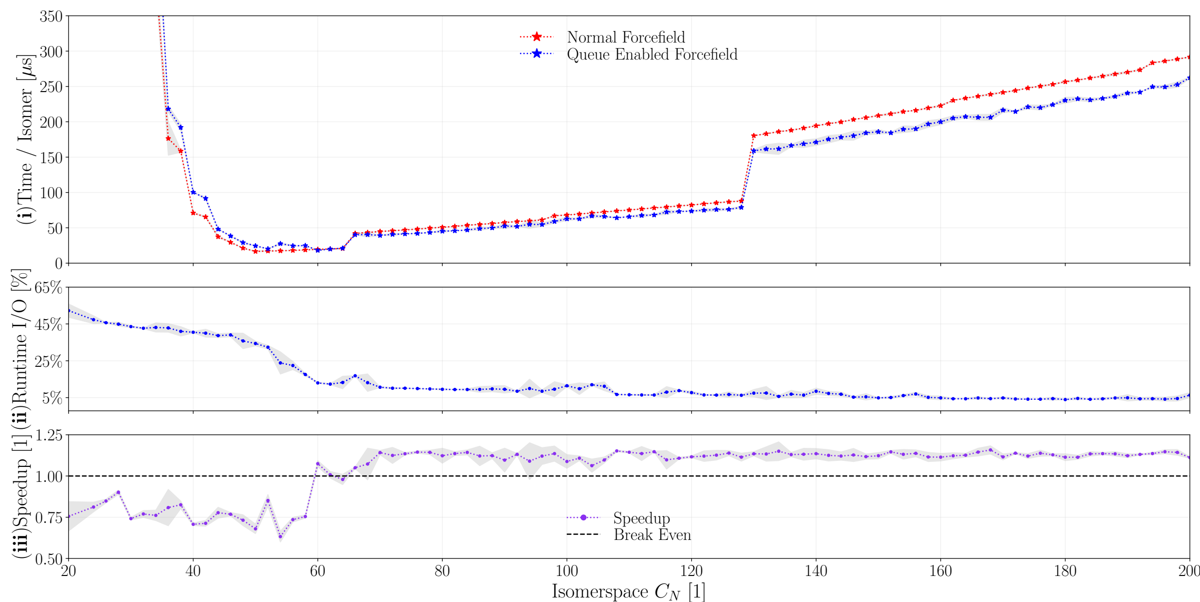


Figure 4.15: *(**i**) Forcefield optimisation time per isomer as a function of isomerspace C_N for the parallel CUDA implementation and the parallel CUDA implementation using the `IsomerQueue`. (**ii**) Fraction of the runtime spent on `IsomerQueue` operations. (**iii**) Speedup of the `IsomerQueue` implementation relative to the previous implementation.*

As we can see from Fig. 4.15 the queue enabled design, provides only a subtle performance gain of approximately $1.15\times$ in the isomerspace range $\{C_{70}, \cdots, C_{200}\}$ over the previous design. Notably it takes a while for this design to reach its full potential, we see that prior to C_{60} the queue enabled design is actually slower than the previous design. The reasons are twofold, firstly the queue based design relies on the `IsomerBatch` always being full of `NOT_CONVERGED` isomers to achieve maximum performance, and for small isomerspaces this is not the case. Secondly, the queue based design requires some overhead to manage the queue, and this overhead is not completely negligible, as we can see from Fig. 4.15 (**iii**). Fortunately the overhead scales linearly with the number of isomers in the queue, and so as the isomerspace increases the overhead becomes a smaller and smaller fraction of the total runtime.

Moreover, the performance of the queue enabled design is directly proportional to the convergence properties that our optimisation algorithm exhibits, this is perhaps the greatest strength of the queue enabled design. If we were to alter the optimisation algorithm or change the convergence criterion we would expect the performance of the queue enabled design to change accordingly.

We have yet to exhaust all performance optimisation avenues, we will explore that which remains in the next subsection.

4.6.3 Fine-Tuning the Forcefield Optimisation Kernel

Per the discussion in chapter 2.1.1 of discrete resource allocation on Nvidia GPUs, we ought to investigate the impact of restricting register allocation on occupancy and memory traffic and consequently performance. We know from figure Fig. 3.6 that only specific intervals of maximum registers per thread need be tested. With this in mind we proceed to benchmark the forcefield optimisation with the relevant different `-maxrregcount` arguments.

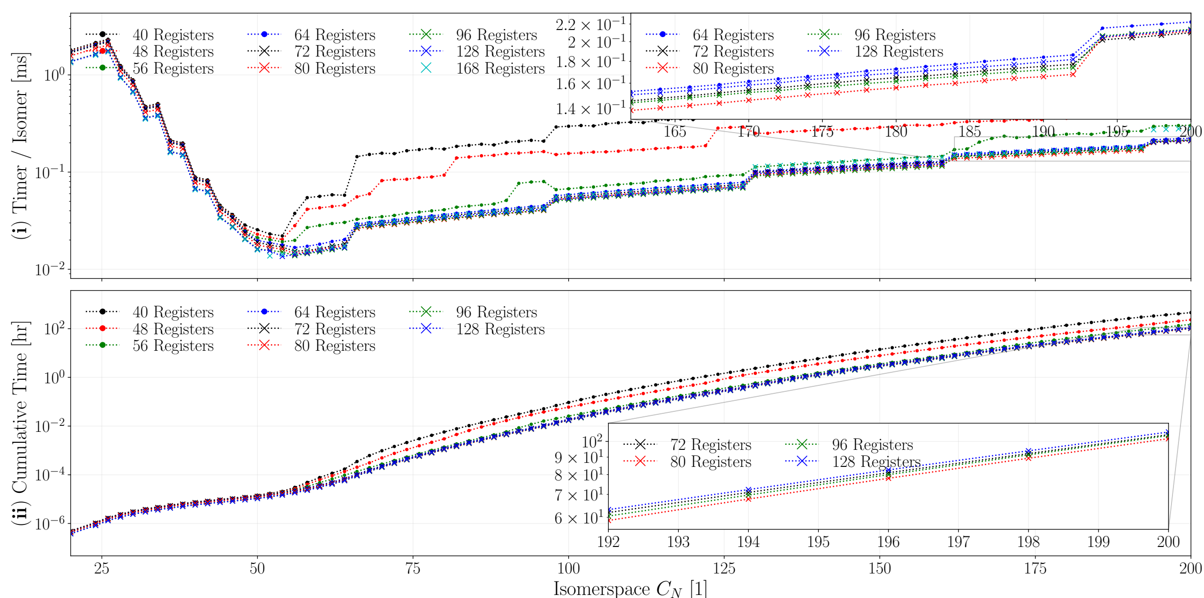


Figure 4.16: *(**i**) Optimisation time per isomer as a function of isomerspace C_N shown for all significant settings of* `-maxrregcount`*. (**ii**) Predicted cumulative hours of optimisation required up to optimise all isomers up to a given isomerspace C_N, tested for the same settings of* `-maxrregcount`*. Error-bars have been omitted from this plot due to cluttering concerns.*

What we discover from Fig. 4.16 is that the optimal register allocation per thread varies somewhat with isomerspace. In the interval $[C_{20}, C_{52}]$ 168 registers per thread wins out, this actually makes perfect sense as this is the domain prior to full saturation of the GPU (not enough isomers in these isomerspaces to saturate the GPU) and so we would expect high register counts to outperform. Then for C_{54} 128 registers is ideal, and in the interval $[C_{56}, C_{64}]$ 96 registers is optimal. Then for $[C_{66}, C_{96}]$ 80 registers is ideal. For the interval $[C_{98}, C_{128}]$ 64 registers is optimal. Finally, for $[C_{130}, C_{160}]$ 96 registers is once again ideal. In the penultimate stretch of isomerspaces $[C_{162}, C_{192}]$ 80 registers outperforms. And Finally for $[C_{194}, C_{200}]$ 72 registers wins. We can see that the optimal register allocation varies, but this plot does not immediately yield any insight into what happen when we vary the register count.

In order to perform more detailed analysis of the kernel we must reduce the dimensionality of the problem. We will do this by fixing the isomerspace and varying the register count. As per figure Fig. 3.6 we note that a block size of 128 threads provides

maximum occupancy for any given register count. We will therefore fix the block size / isomerspace to 128 threads.

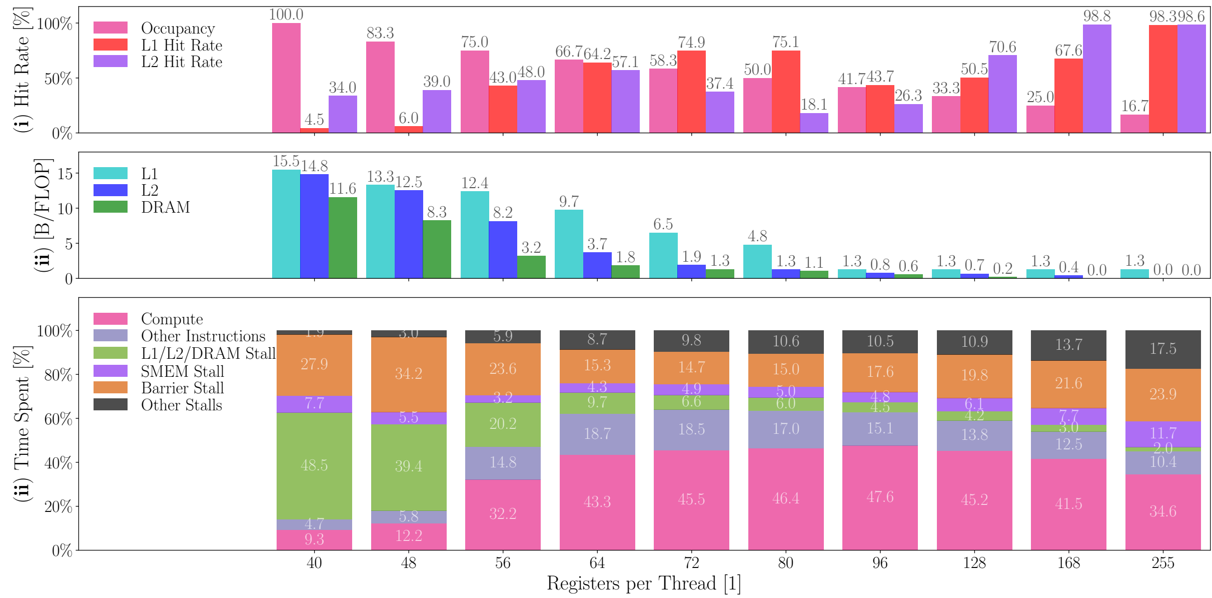


Figure 4.17: *(**i**) Hit rate of the L1 and L2 cache as well as occupancy as a function of register count. (**ii**) reciprocal arithmetic intensity to each level of the memory hierarchy. (**iii**) Fractional time spent segmented into various warp states.*

What we see in Fig. 4.17 some expected trends in subplots (**i**) and (**ii**), namely that as the amount of resources allocated to each thread increases the memory traffic to L1, L2 and most importantly DRAM significantly decreases, and as a result cache hit rates improve as more of the relevant spills are resident in cache at any time. The fact that memory traffic to DRAM and L2 drops all the way to $0 Bytes/FLOP$ shows that the kernel by design can be executed entirely in registers and shared memory, given enough resources. One might erroneously think that 72 registers per thread is the optimal register count looking at subplot (**iii**), here we see that warps are spending 64% of the time issuing instructions, but in fact we are only interested in the time spent issuing compute instructions as the amount of floating point instructions that need to be issued is an invariant of the problem, thus maximizing the amount of time spent issuing compute instructions is the only thing which has a 1:1 correlation with time spent in the kernel. For this purpose we see that 96 registers is ideal, corroborating our findings in Fig. 4.16. Fig. 4.17 also answers the question of why simply allocating maximal number of registers is not the best strategy: as the register count exceeds 96 we observe three trends, **1**) the fractional time spent in the 'other stalls' state increases as things like waiting for arithmetic pipes to clear and waiting for the results of a previous arithmetic operation become visible as occupancy decreases. **2**) More time is spent waiting for barriers to resolve and finally **3**) stalls due to waiting for shared memory transactions to finish increase, these exist primarily because of unavoidable bank conflicts in the memory access pattern. We suggest that all three trends are a display of the magnification effect, that is all these stalls were present previously, but other issues were more prominent. To further solidify this point, note that there is a caveat to all

these performance counter statistics, NVIDIA's profiling tools used to only increment a stall counters when the warp scheduler had no eligible warps for a given cycle. Since compute capability 3.0 however, each stalled warp increments its 'most critical' stall reason by one every cycle.[2] That is to say we do not actually know for a fact that the most represented stall reason is actually what causes the warp scheduler to lose out on issuing instructions. We simply assume it to be the case.

In Fig. 3.13 we actually did not see any significant improvement in performance when using the warp shuffle based reduction, in fact for isomerspaces $[C_{128}, C_{256}]$ performance was equal to or worse than tree based no-conflict reduction. What we notice from our analysis of the kernel performance is however that barrier stalls are a not insignificant portion of the stall reasons, so we will attempt to reduce the number of barriers by using a warp shuffle based reduction.

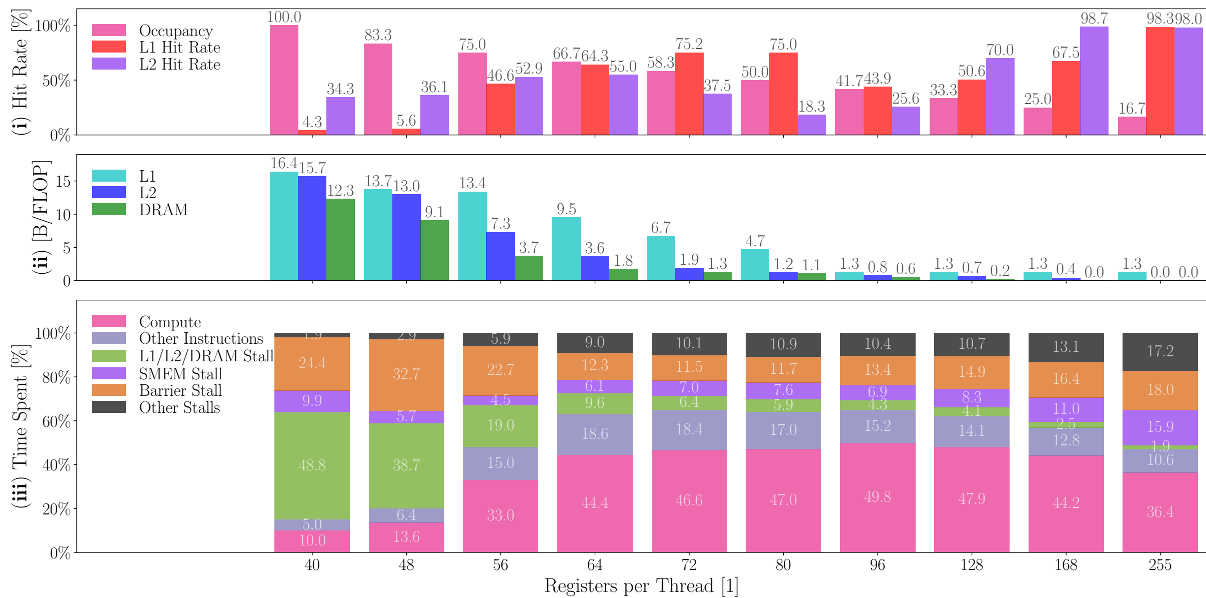


Figure 4.18: *(**i**) Hit rate of the L1 and L2 cache as well as occupancy as a function of register count. (**ii**) reciprocal arithmetic intensity to each level of the memory hierarchy. (**iii**) Fractional time spent segmented into various warp states.*

The trends are much the same as in Fig. 4.17, but importantly we notice that the compute fractions have all increased by a small amount, for 96 registers it increased from 47.6% to 49.8% a 4.6% relative improvement that we expect translates to a 4.6% relative improvement in performance. We note that the performance gain appears to have come from a reduction in barrier stalls and time spent waiting for shared memory transactions, this is extremely reasonable as the warp shuffle based reduction requires fewer transactions and has fewer barriers.

The combined effort of carefully choosing register count and analysing the effects of the compound kernel + warp shuffle based reduction, yields some very decent performance gains. The benchmark results are shown in Fig. 4.19. We see that for isomerspaces $[C_{20}, C_{52}]$ the new implementation is roughly 20% worse, this is completely

expected since for small isomerspaces there are not enough isomers to saturate the GPU, so allocating more resources to each isomers gives obvious benefits. For isomerspaces $C_{66}, \cdots, C_{200}$ the program is anywhere from 20% to 80% faster than the prior implementation, this is the main takeaway.

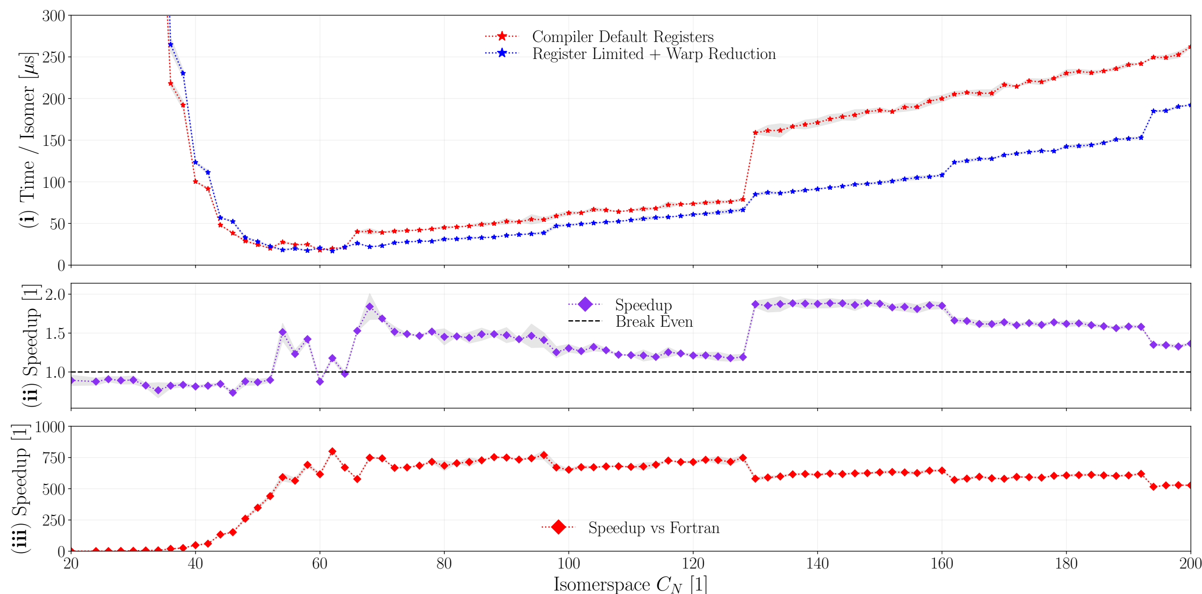


Figure 4.19: *(**i**) Time per isomer as a function of isomerspace C_N comparing the queue implementation from Fig. 4.15 with one where we employ* `--maxrregcount 80` *and a warp shuffle based reduction. (**ii**) Shows speedup of the new implementation over the prior implementation. (**iii**) Shows speedup of the new implementation over the Fortran implementation.*

In Fig. 4.19 (**iii**) we see that forcefield optimisation is now between 750 and 500 times faster than the Fortran implementation in the isomerspace range $\{C_{62}, \cdots, C_{200}\}$. In the next chapter we will investigate the performance of the entire pipeline and opportunities for parallelisation.

4.7 Flatness

In the preceding sections we have shown how to parallelise and implement a forcefield optimisation efficiently, we have shown that the forcefield optimisation yields more accurate results than the previous state of the art forcefield by Wirz et al.[18] (Fig. 4.10). The forcefield is not perfect however, specifically the forcefield tends to occasionally produce concave structures or concave sub-structures (regions of the fullerene). In this section we will explore how flatness can be used to improve the forcefield.

4.7.1 Introduction

In chemistry aromaticity is a property of cyclic or ring-like molecules or regions of molecules. An aromatic molecule is one where electrons from π-orbitals are shared between all atoms of the structure. If one considers the analogy of the benzene molecule (Fig. 4.20) one can conceptually think of the pi-orbital electrons moving around, above and below, the benzene ring pushing it from either side, consequently giving rise to planarity.

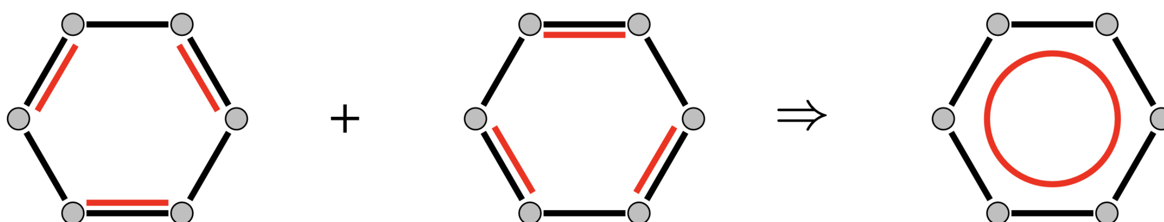


Figure 4.20: *The aromatic benzene molecule. Carbon atoms (**gray**) with σ-bonds (**black**) and free π-orbital bonds (**red**).*[12]

Pedersen anecdotally measured the planarity of the faces of the $C_{60} - I_h$ (icosahedral) fullerene isomer, optimised using DFT methods, finding that pentagons were exactly planar and that the mean distance to the least square plane was of order $10^{-6}\mathring{A}$ for hexagons.[12] It was therefore posited that planarity or flatness, as we will sometimes call it to distinguish it from graph planarity, might be an important property to consider when optimising the geometry of a fullerene. Let us now explore how we can compute the flatness of the faces within a fullerene.

4.7.2 Computing Flatness

The process of computing the flatness of molecular region can be decomposed into a problem of finding the best plane for a set of points, in our case coordinates of atoms within a given hexagon / pentagon.

A plane in $\mathbb{R}^3$ can be represented by a normal vector $\boldsymbol{n}$, and a point in the plane $\boldsymbol{x}_0$. If we perform a coordinate transformation such that $\boldsymbol{x}_1 + \cdots + \boldsymbol{x}_d = \boldsymbol{0}$ (d is the face-degree) then the plane equation reads (Eq. (4.2)):

$$\tilde{\boldsymbol{X}}\boldsymbol{n} = \boldsymbol{0} \tag{4.2}$$

Where $\tilde{\boldsymbol{X}}$ is a matrix of the transformed coordinates of the atoms in the hexagon/pentagon face. Now for sets with more than three points Eq. (4.2) is an overdetermined system of equations, and we can find the normal vector $\boldsymbol{n}$ by solving the least squares problem (Eq. (4.3)).

$$\tilde{\boldsymbol{X}}^T\tilde{\boldsymbol{X}}\boldsymbol{n} = \boldsymbol{0} \tag{4.3}$$

Let $\boldsymbol{A} \equiv \tilde{\boldsymbol{X}}^T\tilde{\boldsymbol{X}}$, now in the case where all the points are within the same plane $\boldsymbol{n}$ is an eigenvector to $\boldsymbol{A}$ with $\lambda = 0$. If the λ has multiplicity 2 then the points are on a line, and if the multiplicity is 3 then the points are all the same.

In the scenario where they do not lie exactly in a plane the solution $\boldsymbol{n}$ to Eq. (4.3) is the eigenvector corresponding to the smallest eigenvalue of $\boldsymbol{A}$, λ_0. Thus, λ_0 is a measure of how flat the face is.

Now from this there is a fairly simple way to incorporate flatness into our forcefield calculation:

$$E_{flat} = \frac{K_{flat}}{2}\sum_{f\in F}\lambda_f \tag{4.4}$$

Where F is the set of faces of the fullerene, and λ_f is the smallest eigenvalue of the least squares problem for the face f. Now the gradient derivation is a little more involved, but we will not go into the details here. The result however can be seen in Eq. (4.5).

$$\frac{\partial E_{flat}}{\partial \boldsymbol{x}_a} = 2K_{flat}\sum_{i=1}^{3}((\boldsymbol{x}_a - \boldsymbol{x}_{c_i})\cdot\boldsymbol{n}_{f_i})\boldsymbol{n}_{f_i}^T \tag{4.5}$$

Where $\boldsymbol{n}_{f_i}$ is the normal vector to the face f_i and $\boldsymbol{x}_a$ is the coordinate of atom a, $\boldsymbol{x}_{c_i}$ is the centroid of the face f_i. Here the sum is over the neighbouring faces of the atom a: F_1, F_2 and F_3 in Fig. 4.3. The derivation was carried out by Avery and can be found in flatness.pdf.[3]

4.7.3 Implementing Flatness

Now in pseudocode the algorithm for computing the flatness gradient contribution for each atom becomes (see Algorithm 18).

In Algorithm 18 we are expressing how the flatness gradient is computed for a single atom. The function GetFace takes in the graph of the fullerene and the atom and

Algorithm 18 Computing Flatness

```
1:  function FLATNESSGRADIENT(X, G)
2:      g_flat                                              ▷ The flatness gradient
3:      for a ∈ {0, ..., N − 1} do                          ▷ For each atom
4:          g_flat[a] ← {0, 0, 0}                           ▷ Initialise the a^th atom's flatness gradient
5:          for j ∈ {0, 1, 2} do                            ▷ For each neighbouring face
6:              d, f ← GetFace(G, a, G[a, j])               ▷ Get the j^th neighbouring face
7:              X                                           ▷ Node coordinates in the face
8:              for i ∈ {0, ··· , d − 1} do
9:                  X[i] ← X[f[i]]                          ▷ Fill in the coordinates
10:             end for
11:             x_c ← (X[f[0]] + X[f[1]] + ··· + X[F[d − 1]])/d   ▷ Compute the centroid
12:             X̃ ← X − x_c                                 ▷ Transform the coordinates
13:             A ← X̃^T X̃                                   ▷ Compute the least squares matrix
14:             λ_0 ← SmallestEigenvalue(A)                 ▷ Find the smallest eigenvalue
15:             n ← Eigenvector(A, λ_0)                     ▷ Compute the eigenvector corresponding to λ_0
16:             g_flat[a] ← g_flat[a] + 2K_flat((x_a − x_c) · n)n^T   ▷ Compute the gradient
17:         end for
18:     end for
19:     return g_flat
20: end function
21:
22: function FLATNESSENERGY(X, G)
23:     E                                                   ▷ Stores energy contributions from each atom
24:     for a ∈ {0, ··· , N − 1} do in lockstep             ▷ For each atom
25:         E_a ← 0                                         ▷ Energy of atom a
26:         for j ∈ {0, 1, 2} do                            ▷ For each neighbouring face
27:             d, F ← GetFace(G, a, G[a, j])               ▷ Get the j^th neighbouring face
28:             X                                           ▷ Node coordinates in the face
29:             for i ∈ {0, ··· , d − 1} do
30:                 X[i] ← X[F[i]]                          ▷ Fill in the coordinates
31:             end for
32:             x_c ← (X[F[0]] + X[F[1]] + ··· + X[F[d − 1]])/d   ▷ Compute the centroid
33:             X̃ ← X − x_c                                 ▷ Transform the coordinates
34:             A ← X̃^T X̃                                   ▷ Compute the least squares matrix
35:             λ_0 ← SmallestEigenvalue(A)                 ▷ Find the smallest eigenvalue
36:             E[a] ← E[a] + (K_flat / 2d) λ_0^2           ▷ Compute the energy contribution
37:         end for
38:     end for
39:     return reduce(E, +)
40: end function
```

the face index and returns the degree of the face and the face itself. The function SmallestEigenvalue takes in a matrix and returns the smallest eigenvalue of the matrix, and Eigenvector takes in a matrix and an eigenvalue and returns the corresponding eigenvector. A custom 3x3 symmetric matrix structure was written (`SymMat3` in coord3d.cuh) to store the upper triangular elements of the matrix, and to facilitate eigenvalue and eigenvector computation in closed form.

This was implemented and sparsely benchmarked to find that performance had degraded by approximately a factor of 6. We believe this performance degradation is a result of two issues:

1. **Excessive Computation:** Every thread needs to compute the least squares matrix for each face and finally solve the eigenvalue problem. This means that each least square matrix is computed and solved 5 or 6 times for each atom.
2. **Increased Register Pressure:** Each thread now has to store all neighbours in its neighbouring faces, the coordinates of the nodes in the face and the least square matrix.

This is what motivated us to look for a solution that does not require us to recompute the least squares matrix for each face.

4.7.4 Optimisation: Solve the Eigenvalue Problem Once Per Face

If we dissect Algorithm 18 we see that we require the following information in order to compute the flatness gradient with respect to an atom:,

1. The normal vector to each of its neighbouring faces $\boldsymbol{n}_{f_i}$
2. The centroid of each of its neighbouring faces $\boldsymbol{x}_{c_i}$
3. Its own position $\boldsymbol{x}_a$

So if we are somehow able to compute (in parallel) and store all the N_f normal vectors and centroids in shared memory, then the gradient computation becomes far more efficient. This requires a little of preliminary work, since we have so far been working with the cubic-graph in a vacuum.

Specifically we need to:

- Assign unique indices to each face in the fullerene.
- Store the indices of the neighbouring faces that each atom has.
- For thread $i \in [0, N_f[$ we need to store the cubic-indices of nodes that make up the ith face.

Fortunately this is the dualisation problem all over again, and we have already seen how to solve that efficiently in parallel in Algorithm 26. We can use the same approach to solve this problem, so we omit the details here, but know that it occurs during the construction of the `NodeNeighbours` structure.

In Algorithm 19 we have implemented these optimisations, $\boldsymbol{F}[i,0],\cdots,\boldsymbol{F}[i,d-1]$ is the node indices belonging to i^{th} face, and $\boldsymbol{G}^*[a,j]$ is the jth neighbouring face of atom a, not to be confused with the dual graph produced by `BuckyGen`, where $\boldsymbol{G}^*[i,j]$ is the jth neighbouring face of the ith face. With the updated design the gradient computation requires us to store normal vectors and centroids in shared memory ($\boldsymbol{N}$, $\boldsymbol{X_c}$) such that it can be accessed by all threads to compute Eq. (4.5). The energy computation becomes a simple reduction of the energy contributions from each face. The optimised implementation of the flatness energy and gradient computation can be found in forcefield.cu.

4.7.5 Performance Impact of Flatness Term

In Algorithm 19 we have seen how it is possible to incorporate the flatness term efficiently into the forcefield calculations without disrupting the existing method. If the performance penalty for adding a forcefield term is too great then it becomes a lot less appealing to include it, so let us investigate how the flatness enabled forcefield performs in comparison with the CUDA Pedersen implementation.

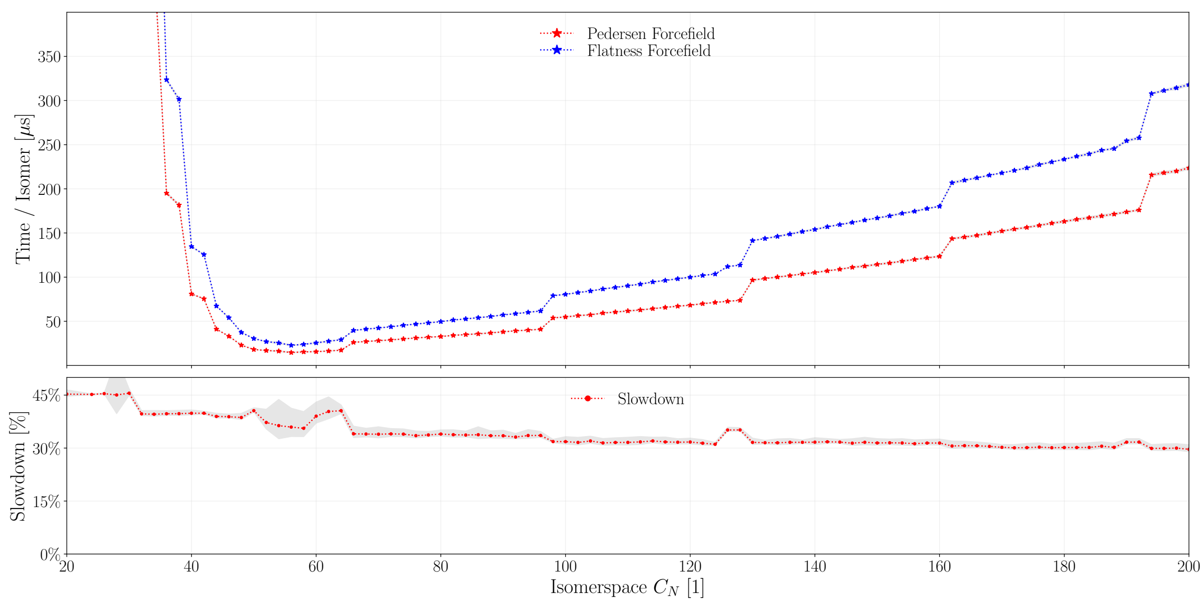


Figure 4.21: *Performance comparison of the CUDA Pedersen forcefield with* **blue** *and without* **red** *the flatness term. We measure here the time (per isomer) that it takes to perform* $5N$ *iterations of forcefield optimisation.*

We see that the flatness enabled forcefield is about 45% slower than the CUDA Pedersen implementation in the early isomerspaces and drops steadily towards 30% slower at C_{200} we presume that this is a matter of different component scaling. The reductions in the forcefield scale with $\mathcal{O}(N\log_2(N))$, whereas the energy and gradient calculations scale with $\mathcal{O}(N)$. Whether we are willing to sacrifice the 30% performance hit for the

Algorithm 19 Optimised Flatness Energy and Gradient Computation

function FLATNESSGRADIENT($\boldsymbol{X}, \boldsymbol{G}, \boldsymbol{G}^*, \boldsymbol{F}$) ▷ Coordinates, graph, dual graph, faces
  $\boldsymbol{N}, \boldsymbol{X_c}$ ▷ Normal vectors and centroids
  for $i \in \{0, \dots, N_f - 1\}$ **do in lockstep** ▷ For each face
    $\boldsymbol{X}$ ▷ Node coordinates in the face
    for $j \in \{0, \cdots, d-1\}$ **do**
      $\boldsymbol{X}[j] \leftarrow \boldsymbol{X}[\boldsymbol{F}[i, j]]$ ▷ Fill in the coordinates
    end for
    $\boldsymbol{X_c}[i] \leftarrow (\boldsymbol{X}[\boldsymbol{F}[i,0]] + \boldsymbol{X}[\boldsymbol{F}[i,1]] + \cdots + \boldsymbol{X}[\boldsymbol{F}[i, d-1]])/d$ ▷ Compute the centroid
    $\tilde{\boldsymbol{X}} \leftarrow \boldsymbol{X} - \boldsymbol{X_c}[i]$ ▷ Transform the coordinates
    $\boldsymbol{A} \leftarrow \tilde{\boldsymbol{X}}^T \tilde{\boldsymbol{X}}$ ▷ Compute the least squares matrix
    $\lambda_0 \leftarrow \mathsf{SmallestEigenvalue}(\boldsymbol{A})$ ▷ Find the smallest eigenvalue
    $\boldsymbol{N}[i] \leftarrow \mathsf{Eigenvector}(\boldsymbol{A}, \lambda_0)$ ▷ Find the corresponding eigenvector
  end for
  $\boldsymbol{g_{flat}}$ ▷ The flatness gradient
  for $a \in \{0, \dots, N-1\}$ **do in lockstep** ▷ For each atom
    $\boldsymbol{g_{flat}}[a] \leftarrow \{0, 0, 0\}$ ▷ Initialise the ath atom's flatness gradient
    for $j \in \{0, 1, 2\}$ **do** ▷ For each neighbouring face
      $i \leftarrow \boldsymbol{G}^*[a, j]$ ▷ Get the face index
      $\boldsymbol{g_{flat}}[a] \leftarrow \boldsymbol{g_{flat}}[a] + 2K_{flat}((\boldsymbol{x}_a - \boldsymbol{X_c}[i]) \cdot \boldsymbol{N}[i])\boldsymbol{N}[i]^T$ ▷ Compute the gradient
    end for
  end for
  return $\boldsymbol{g_{flat}}$
end function

function FLATNESSENERGY($\boldsymbol{X}, \boldsymbol{G}, \boldsymbol{G}^*, \boldsymbol{F}$) ▷ Coordinates, graph, dual graph, faces
  $\boldsymbol{E}$ ▷ Stores the energy contributions from each face
  for $i \in \{0, \dots, N_f - 1\}$ **do in lockstep** ▷ For each face
    $\boldsymbol{X}$ ▷ Node coordinates in the face
    for $j \in \{0, \cdots, d-1\}$ **do**
      $\boldsymbol{X}[j] \leftarrow \boldsymbol{X}[\boldsymbol{F}[i, j]]$ ▷ Fill in the coordinates
    end for
    $\boldsymbol{X_c}[i] \leftarrow (\boldsymbol{X}[\boldsymbol{F}[i,0]] + \boldsymbol{X}[\boldsymbol{F}[i,1]] + \cdots + \boldsymbol{X}[\boldsymbol{F}[i, d-1]])/d$ ▷ Compute the centroid
    $\tilde{\boldsymbol{X}} \leftarrow \boldsymbol{X} - \boldsymbol{X_c}[i]$ ▷ Transform the coordinates
    $\boldsymbol{A} \leftarrow \tilde{\boldsymbol{X}}^T \tilde{\boldsymbol{X}}$ ▷ Compute the least squares matrix
    $\lambda_0 \leftarrow \mathsf{SmallestEigenvalue}(\boldsymbol{A})$ ▷ Find the smallest eigenvalue
    $\boldsymbol{E}[i] \leftarrow \frac{K_{flat}}{2} \lambda_0^2$ ▷ Compute the energy
  end for
  return $\mathsf{reduce}(\boldsymbol{E}, +)$
end function

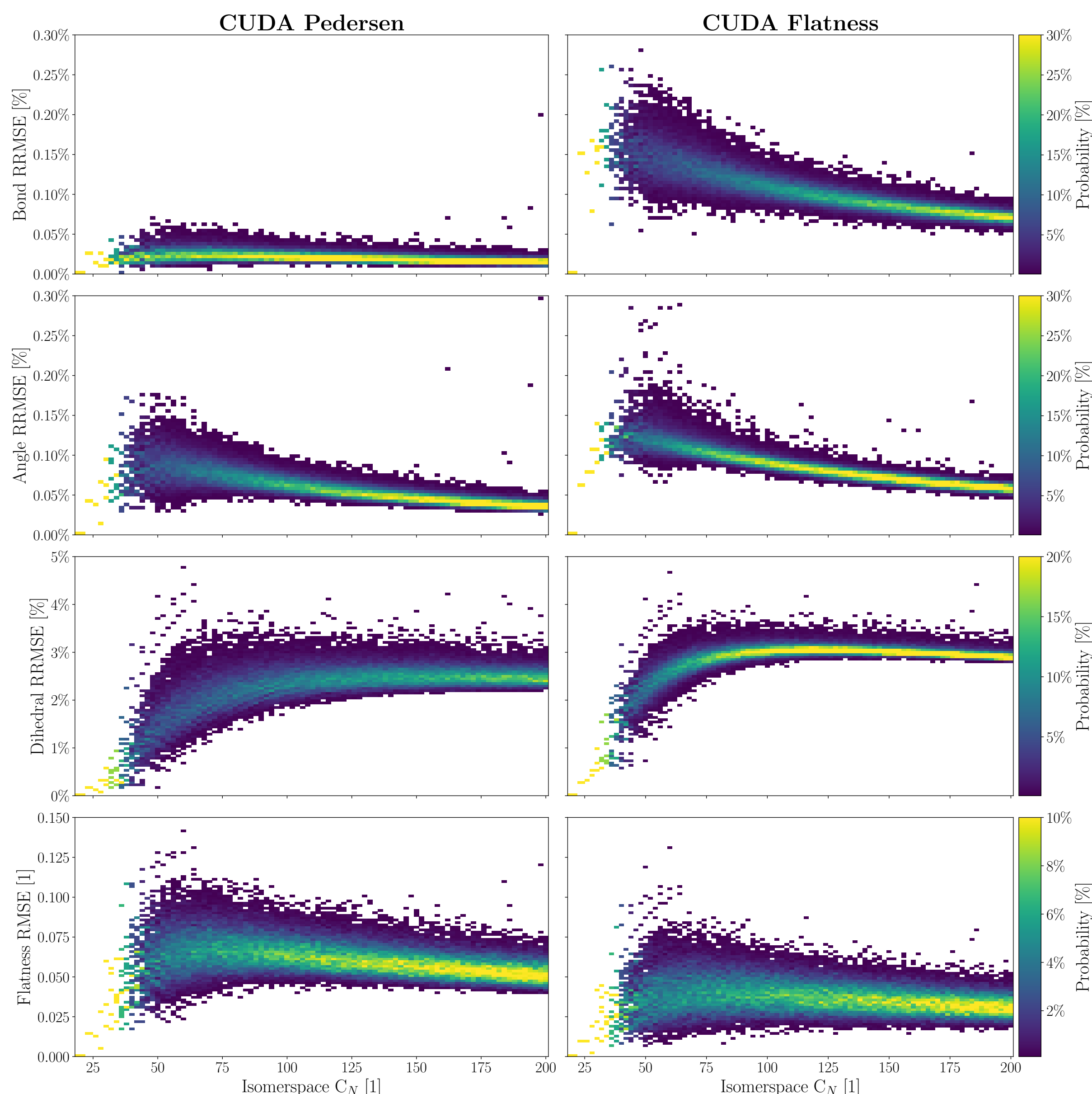


Figure 4.22: *RRMSE of bond lengths, angles, dihedrals and RMSE of flatness for CUDA Pedersen forcefield and CUDA flatness forcefield. Each plot contains a histogram of RRMSE/RMSE values for each isomerspace C_N, the probability is the fraction of isomers in an isomerspace that fall within a given RRMSE/RMSE bin. Bins were linearly distributed from the minimum to the maximum value of the y-axes in each plot.*

flatness term is a matter of quality of results versus performance. We should note that methods with fewer floating point operations exist for computing the eigenvalues of 3x3 symmetric matrices, but the method used here is numerically stable.

4.7.6 Results

With flatness efficiently implemented let us investigate the impact it has on fullerene internal coordinates, bond-lengths, bond-angles, dihedral-angles and finally flatness. We use the same methodology as in Section 4.4 to perform the quantitative comparison of the flatness enabled forcefield with the CUDA Pedersen implementation.

Note that for flatness the measure is RMSE rather than RRMSE, since flatness is an absolute value rather than a relative value. The equilibrium parameter for flatness is 0 and as such it is not possible to compute RRMSE. We see that the flatness enabled forcefield performs better at flatness than the CUDA Pedersen forcefield, and it appears to do this by relaxing the constraint on angles and especially bond-lengths. Qualitatively we inspected some of the isomers that improved the most, with regard to RMSE flatness, these isomers are shown in Fig. 4.23.

What might be visually apparent from Fig. 4.23 is that the flatness enabled forcefield seems to produce slightly more convex structures by 'inflating' the molecules. As we saw from the quantitative analysis this is achieved through slight relaxation of the bond-lengths.

4.7.7 Discussion

We have seen how the flatness calculation can be integrated efficiently into the CUDA forcefield calculations, incurring a mere 30% performance deficit. We have also seen that the flatness enabled forcefield performs better than the CUDA Pedersen forcefield in terms of flatness, at the cost of slightly higher bond-length and angle deviation. Since the forcefield calculation relies on parametrization of ideal bond-lengths, angles et cetera, we ultimately cannot say whether the flatness enabled forcefield is more accurate than the CUDA Pedersen forcefield. To get a measure of whether it is more accurate we would have to compare these structures to experimental data, or practically speaking, DFT optimised structures. This assessment is beyond the scope of this thesis, but it needs to be investigated in the future.

One caveat to the energy and gradient flatness calculations that has been omitted thus far, is that the force constant K_{flat} has not been determined yet, force constants are determined using DFT methods, so this is a task for the future. In out implementation we have set $K_{flat} = K_{bond}$ through lack of a better value. Moreover, the gradient calculation is currently the gradient w.r.t. λ_0, not λ_0^2, it has not been decided whether the flatness term should be harmonic in λ_0 or $\sqrt{\lambda_0} = \sigma_0$ (Singular Value Decomposition of $\tilde{\boldsymbol{X}}$).

4.8 Summary

In Section 4.2 we go through the every component of the forcefield optimisation algorithm, Algorithm 9, Algorithm 12, Algorithm 14 and transform each of these into algorithms which fit the lockstep paradigm: Algorithm 10, Algorithm 13, Algorithm 15 and Algorithm 11.

In Section 4.3 we outline the data structure hierarchy of the CUDA implementation of the forcefield optimisation Fig. 4.5 and introduce some key components of the CUDA/C++ framework developed in this thesis like the `IsomerBatch` and the operations defined on it. Moreover, the `LaunchCtx` is introduced as a way of controlling the execution of CUDA kernels. We provide a small example demonstrating how the framework

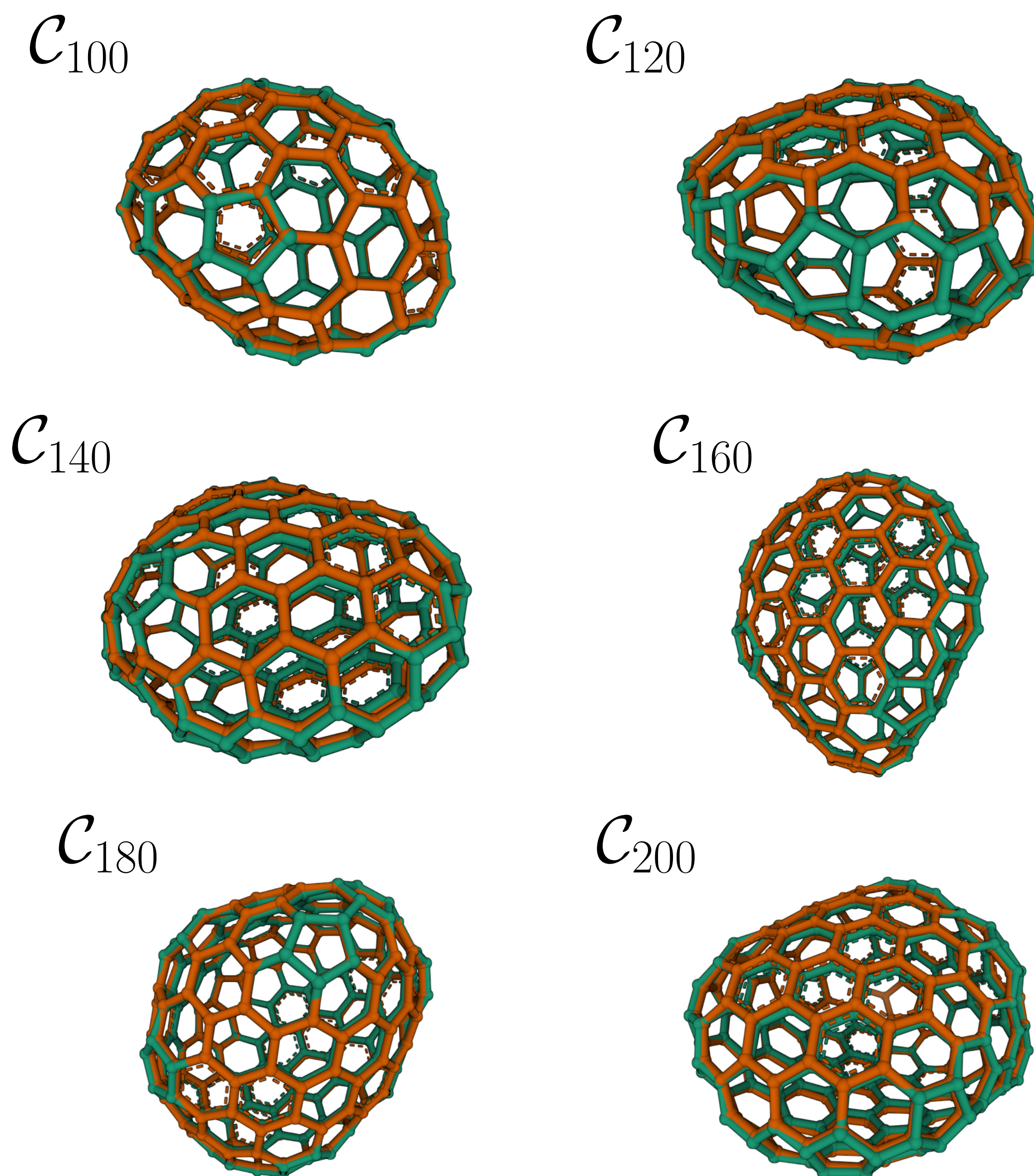


Figure 4.23: *Select isomers from our random samples of isomerspaces $C_{100}, C_{120}, \cdots, C_{200}$ that improved the most in terms of RMSE flatness. The CUDA Pedersen optimised molecules are drawn in* **teal** *with the flatness optimised molecules drawn in* **orange**. *These visualisations were made using the Visual Studio Code extension Protein Viewer.[16] Dashed lines indicate that planarity of a face fell within a default threshold.*

incorporates into standard C++ code (Listing 4.1).

In Section 4.4 we saw how the CUDA/C++ implementation produces geometries that are in better agreement with the forcefield parameters than the Wirz et al. implementation in the *fullerene* program (Fig. 4.9 and Fig. 4.10). We further performed an analysis of the convergence statistics of our randomly sampled isomers (Fig. 4.11) we found that the variation is small, yet if we want to ensure that 99.8% of the isomers converge we are required to perform $5N$ iterations Fig. 4.12.

Motivated by the fact that we would potentially be wasting 40% more iterations than required on average (Fig. 4.12) if we use fixed number of iterations for all isomers, we introduced a queue based approach in Section 4.5.6. We go through each of its natural sequential host-side operations, `push` and `pop`, as well as the collective operations `refill_batch` and `drain_batch`. We provide pseudocode that explains how one can implement these collective operations (Algorithm 16 and Algorithm 17). Finally, we benchmarked the queue based implementation and saw a modest 15% improvement in performance (Fig. 4.15). We went on to optimise this further by reducing the number of registers allocated to each thread and substituting our prior reduction method with a warp-based reduction method, yielding yet another 20-80% performance improvement (Fig. 4.19). We compared the optimised CUDA/C++ implementation to the Wirz et al. implementation in the *fullerene* program and found that the CUDA/C++ implementation is 450-700x faster (Fig. 4.19).

In Section 4.7 we introduced the concept of flatness and showed how it can be implemented naively in the CUDA/C++ forcefield (Algorithm 18) but better yet, how we can optimised on these algorithms to reduce duplicate computation (Algorithm 19). We analysed the resulting geometries and the RRMSE bond-length, angle, dihedral and RMSE flatness distributions for randomly sampled isomers (Fig. 4.22). We found that the flatness enabled forcefield yields more convex structures, owing to the face planarity, yet sacrifices bond-length and angle accuracy RRMSE. Our performance measurements showed that the efficient addition of flatness calculations to the CUDA/C++ forcefield implementation incurs a mere 30% performance deficit (Fig. 4.21) compared to the anecdotally assessed degradation of 600% for the naive implementation.

Chapter 5

Fully Lockstep Parallel Pipeline

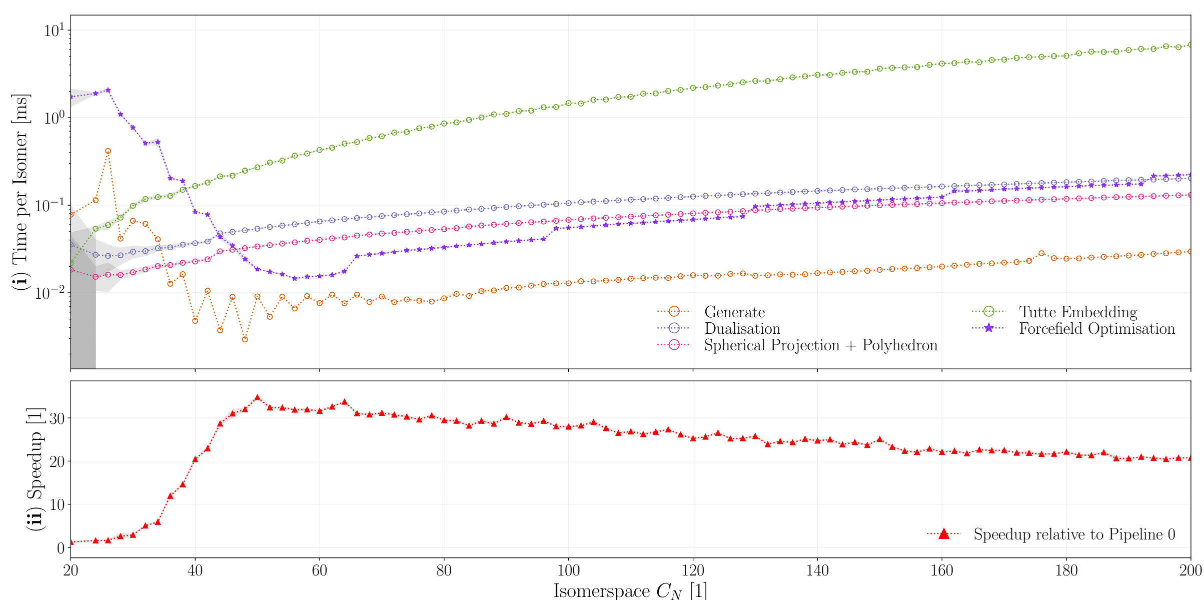


Figure 5.1: *Time per Isomer performance for lockstep parallel forcefield optimisation ($\star$), sequential Tutte embedding ($\circ$), sequential spherical projection ($\circ$), sequential dualization from dual to cubic representation ($\circ$) and `BuckyGen` generation of graphs ($\circ$). The shaded areas represent performance ± 2 standard deviations. In* ***(i)*** *the y-axis is logarithmic to make faster components visible. $\star$ marker is used to indicate a parallel component, $\circ$ marker for sequential components.*

In the preceding chapter, we have shown that not only is the forcefield massively data parallel and possible to parallelize, but it is possible to write the implementation in such a way that 50% of the runtime can be spent on pure floating-point arithmetic Fig. 4.19, a remarkable result. Furthermore, we have shown, it is possible through the ingenuity of the `IsomerQueue` (Section 4.5), an efficient and completely novel, flexible lockstep parallel queue implementation that fits in the SIMT model, to allow isomers to converge at different rates (Section 4.4.2). We may now ask ourselves what is next, well if we plot out the runtime of the entire pipeline from `BuckyGen` to the forcefield optimisation after parallelizing the forcefield we arrive at Fig. 5.1. In this figure, we see that the forcefield is now so fast that it beats components of the pipeline that were previously trivial.

In this chapter, we will explore the bottlenecks present in the isomerspace optimisation pipeline Fig. 4.8, systematically investigate each bottleneck, explore to what extent if at all, it is possible to exploit data parallelism and move them into the lockstep-parallel paradigm. Just how much faster can we make these processes? This is the question we will answer here.

5.1 Pipeline 3: Tutte Embedding

Pipeline 1 and its constituents were benchmarked and shown in Fig. 5.1, as discussed in the previous chapter, the parallel forcefield has enabled us to produce a pipeline that is between 20 and 13 times faster, in the $[C_{60}, C_{200}]$ range, than Pipeline 0. What is

immediately apparent through component benchmarks is that Tutte embedding is now responsible for the vast majority of the runtime. From visual inspection of Fig. 5.1 alone we can see that it is an order of magnitude slower that all other components at C_{200}. This is what motivates us to investigate the Tutte embedding process in more detail, just how parallelisable is it?

5.1.1 Tutte Embedding

Given a graph $\boldsymbol{G}$ which is planar and 3-connected, there exists a unique crossing-free embedding with the property that each vertex $v \in \mathcal{V}$ is at the average position of its neighbours. Furthermore, the embedding defines an outer face that bounds the remaining vertices, this face is a convex polyhedron of degree h. We can represent the condition that a vertex v be at the centre of its neighbours with two linear equations, one for the x coordinate and one for the y coordinate. In total for a graph $\boldsymbol{G}$ with N vertices we get $2(N-h)$ equations with $2(N-h)$. Indeed, this system of equations was shown to be non-degenerate in the 3-connected case. Thus solving this system of equations yields a unique solution and planar embedding.[17]

5.1.2 Sequential Tutte Algorithm

Algorithm 20 Sequential Tutte Embedding Algorithm

```
1:  function TUTTEEMBEDDING(G)                                  ▷ Graph to embed
2:      X^2D                       ▷ N × 2 array of floats containing 2D coordinates
3:      X^2D_new              ▷ N × 2 next array of floats containing 2D coordinates
4:      F_0 ← GetFace(G, 0, G[0, 0])                             ▷ Get outer face
5:      converged ← False
6:      while not converged do
7:          Δ_max ← 0                    ▷ Store maximum change of solution in iteration
8:          for j ← 0 to N − 1 do
9:              if not F[j] then
10:                 b_c ← (0, 0)                                      ▷ barycentre
11:                 d̄ = 0                                ▷ Mean distance to neighbours
12:                 for k ← 0 to 2 do
13:                     n ← G[j, k]                                ▷ k^th neighbour
14:                     b_c ← b_c + X^2D[n]
15:                     d̄ = d̄ + norm(X^2D[j] − X^2D[n])
16:                 end for
17:                 b_c ← b_c/3                       ▷ Average position of 3 neighbours
18:                 d̄ ← d̄/3                           ▷ Average distance to 3 neighbours
19:                 X^2D_new[j] ← 0.15 · X^2D[j] + 0.85 · b_c               ▷ Time step
20:                 Δ_rel ← norm(X^2D[j] − X^2D_new[j])/d̄
21:                 Δ_max ← max(Δ_max, Δ_rel)
22:             end if
23:         end for
24:         X^2D ← X^2D_new                                      ▷ Update positions
25:         if Δ_max < tol then                   ▷ Check if change less than tolerance
26:             converged ← True
27:         end if
28:     end while
29:     return X^2D                                        ▷ Return 2D Embedding
30: end function
```

The linear system of equations defined by the Tutte embedding is incredibly sparse specifically $\frac{3}{N-h}$, and while it can be solved by many more sophisticated solvers we present here a simple iterative solver.

First, we must define an outer face, we choose the one represented by the arc $\boldsymbol{G}[0,0] \rightarrow \boldsymbol{G}[0,1]$. The coordinates of these vertices are fixed, and we proceed to solve the system by iteratively updating the position of the *free* vertices. In each iteration, we find the barycentre b_c and the average distance to neighbours $\bar{d}$ for all vertices. Once we have found the barycentres we can update the new positions and store them in a temporary array, $\boldsymbol{X^{2D}_{new}}[j] \leftarrow 0.15 \cdot \boldsymbol{X^{2D}}[j] + 0.85 \cdot b_c$. The double-buffered nature of having both $\boldsymbol{X^{2D}_{new}}$ and $\boldsymbol{X^{2D}}$ is what saves us from potential loop carried dependence. After each iteration we copy the new positions $\boldsymbol{X^{2D}} = \boldsymbol{X^{2D}_{new}}$. We then gauge the solution by finding the maximum relative change Δ_{max} w.r.t. mean neighbour distance $\bar{d}$, if this change is below a certain threshold tol the solution has converged.

5.1.3 Parallel Tutte Algorithm

Fortunately, iterative solvers expose far better parallelism than direct solvers like Gaussian elimination where each row operation is dependent on the previous. Algorithm 20 then lends itself very well to parallelism, indeed most of the algorithm is completely data-parallel.

Algorithm 21 Parallel Tutte Embedding Algorithm

```
1:  function TUTTEEMBEDDING(G_in, X^2D_out)            ▷ Cubic Graph, Output Embeddings
2:    for i ← 0 to M − 1 do in lockstep                 ▷ For each isomer
3:      G ← G_in[i]                                     ▷ Load the j^th graph
4:      F_0 ← GetFace(G, 0, G[0, 0])                    ▷ Define an outer face
5:      for it ∈ {0, . . . 10N − 1} do                  ▷ Perform 10N iterations
6:        Δ_max ← 0                   ▷ Store maximum change of solution in iteration
7:        for j ← 0 to N − 1 do in lockstep             ▷ For each atom
8:          if j ∉ F_0 then                             ▷ If not in outer face
9:            b_c ← (0, 0)                              ▷ barycentre
10:           d̄ = 0                                     ▷ Mean distance to neighbours
11:           for k ← 0 to 2 do                         ▷ For each neighbour
12:             n ← G[j, k]                             ▷ k^th neighbour
13:             b_c ← b_c + X^2D[n]                     ▷ Add neighbour position
14:             d̄ = d̄ + norm(X^2D[j] − X^2D[n])        ▷ Add distance to neighbour
    to d̄
15:           end for
16:           b_c ← b_c/3                               ▷ Mean position of 3 neighbours
17:           d̄ ← d̄/3                                   ▷ Mean distance to 3 neighbours
18:           X^2D_new[j] ← 0.15X^2D[j] + 0.85b_c       ▷ Time step
19:         end if
20:       end for
21:       for j ← 0 to N − 1 do in lockstep
22:         X^2D[j] ← X^2D_new[j]                       ▷ Update positions
23:       end for
24:     end for
25:     X^2D_out[i] ← X^2D                              ▷ Store the j^th embedding
26:   end for
27: end function
```

The parallel algorithm (Algorithm 21) proceeds exactly like the sequential algorithm with few exceptions. Each processing element PE_i is now responsible for computing the barycentre b_c and mean neighbour distance $\bar{d}$ of the jth vertex. Each PE_i then computes the update to the jth position $\boldsymbol{X}^{2D}_{new}[j]$ according to the same formula as per previous (Algorithm 20). Now, notably there is one caveat to lockstep parallelising the Tutte embedding: the convergence criterion checks the maximum relative change Δ_{max} of the solution; and while we could compute this in parallel using a reduction operation, it would break the lockstep parallelism of the algorithm. Therefore, we perform a sufficient (10N was determined) number of iterations as with the forcefield optimisation (Algorithm 10).

5.1.4 Tutte Validation

Algorithm 21 is implemented in CUDA/C++ and found in tutte.cu. We first validate that our lockstep algorithm indeed produces the desired results. We note that floating-point maths is not associative and certain instructions exist on GPUs such as the fused-multiply-add operation, which will produce different yet more accurate results than performing a multiply operation followed by an addition. For these reasons, some discrepancy in the solutions is to be expected.

To validate the parallel algorithm we produce the first 10000 isomers in the isomerspaces $C_{20}, C_{24}, \cdots, C_{200}$ using the `BuckyGen` generator.[5] We then compute the Tutte layout using both the sequential C++ algorithm on a CPU and in parallel using Algorithm 21 on the GPU. For each isomer we compute the average absolute error and average relative error using Eq. (5.1) and Eq. (5.2)

$$\epsilon = |v_{approx} - v_{target}| \tag{5.1}$$

$$\nu = \frac{|v_{approx} - v_{target}|}{|v_{target}| + \delta} \tag{5.2}$$

Here δ is 1 when $v_{target} < 10^{-8}$ and 0 otherwise, this leads to an absolute error calculation when the v_{target} is close to zero. The reason being that comparing floating point numbers that are approximately zero are prone to give large relative errors thus skewing the results. Having computed these relative and absolute errors we bin them in the logarithmic space from 10^{-8} to 10^{-2} and arrive at the probability density maps in Fig. 5.2.

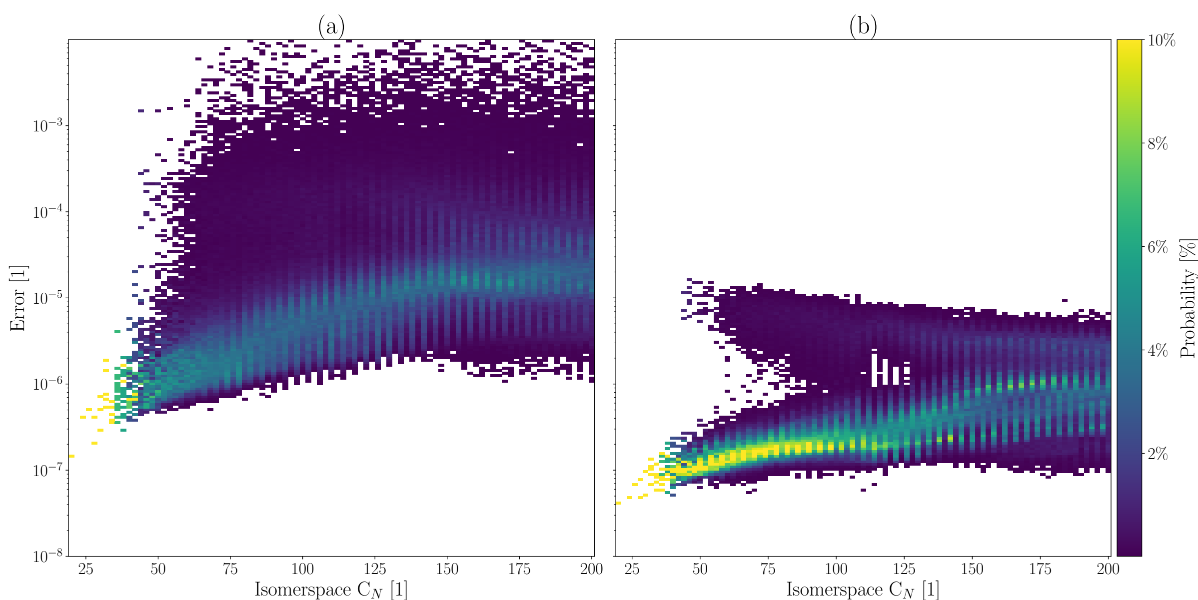


Figure 5.2: *Probability densities of (a) relative error and (b) absolute error based on the first 10000 samples generated by* `BuckyGen` *from each isomerspace* $C_{20}, C_{24}, \cdots, C_{200}$.

We note some curious stripe-like artefacts in Fig. 5.2. This is not entirely unexpected as the `BuckyGen` program produces fullerenes recursively using growth operations from a smaller subset of irreducible fullerenes.[5] This provided the original motivation for producing a set of randomly sampled isomers from each isomerspace as per Algorithm 1.

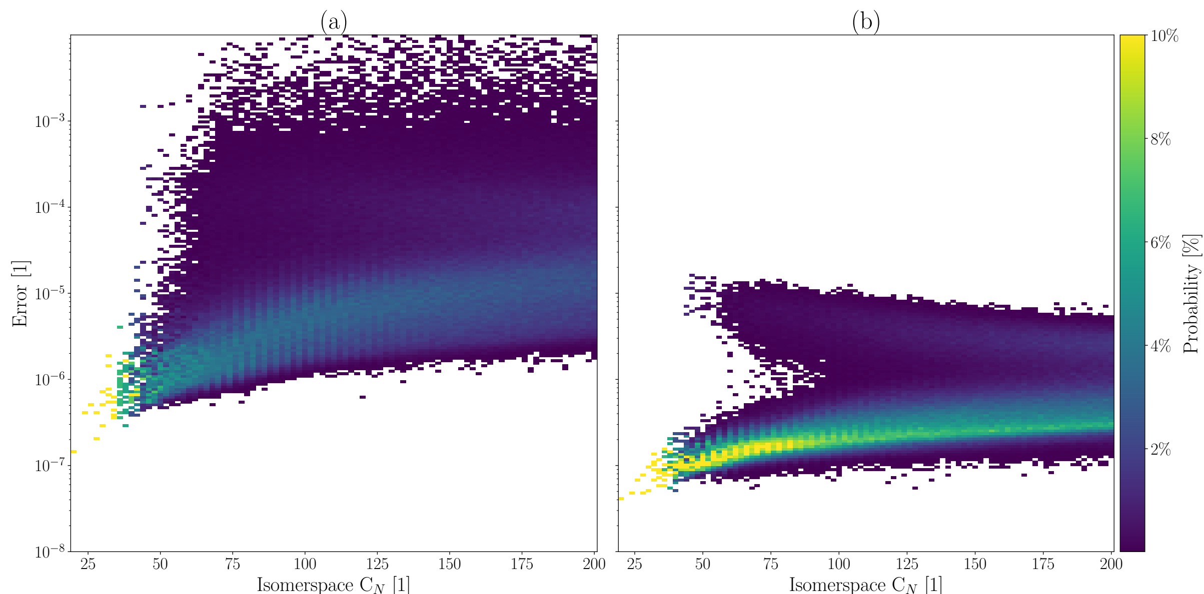


Figure 5.3: *Probability densities of (a) relative error and (b) absolute error based on 10000 randomly sampled isomers, generated using the method described in Algorithm 1, from each isomerspace* $C_{20}, C_{24}, \cdots, C_{200}$.

Now performing the same error analysis using random samples we see that figures Fig. 5.2 and Fig. 5.3 share many of the same features including this double-pronged shape of the absolute error distributions, confirming that this feature is not a product of procedurally generated isomers. Instead, we concede that it has to be a general feature that could warrant future investigation. We suggest here that it could be a feature of the underlying system of equations that has to be solved. The stripe-like artefacts have mostly been resolved, and we arrive at a much smoother and more consistent distribution from sampling the isomerspaces. From Fig. 5.3 we estimate that the vast majority of isomers exhibit relative error in the domain $[10^{-8}, 10^{-3}]$ with the average at around 10^{-6} relative error in the sampled isomerspaces. First and foremost this gives us confidence in the parallel algorithm working correctly and secondly the discrepancy in results is well within reason for the purposes of calculating initial geometry.

5.1.5 *Performance of the Tutte Embedding*

We execute each kernel in sequence (**dualize** → **Tutte** → **projection** → **optimize**). This is repeated 10 times to arrive at mean execution time and standard deviations on the execution time. We compute the optimal batch size using the `LaunchDims` class described previously, this method of course minimizes ghost threads, for a given block size. The kernels are executed on a sample of random isomers picked from the randomly generated isomers (see Algorithm 1). First we measure the performance of the new

parallel Tutte embedding kernel relative to the sequential method, we see from Fig. 5.4 that performance is roughly $\approx$ 1000 times higher for most isomerspaces beyond the saturation point (C_{58}). This demonstrates that the Tutte embedding algorithm fits very well in the lockstep parallel paradigm.

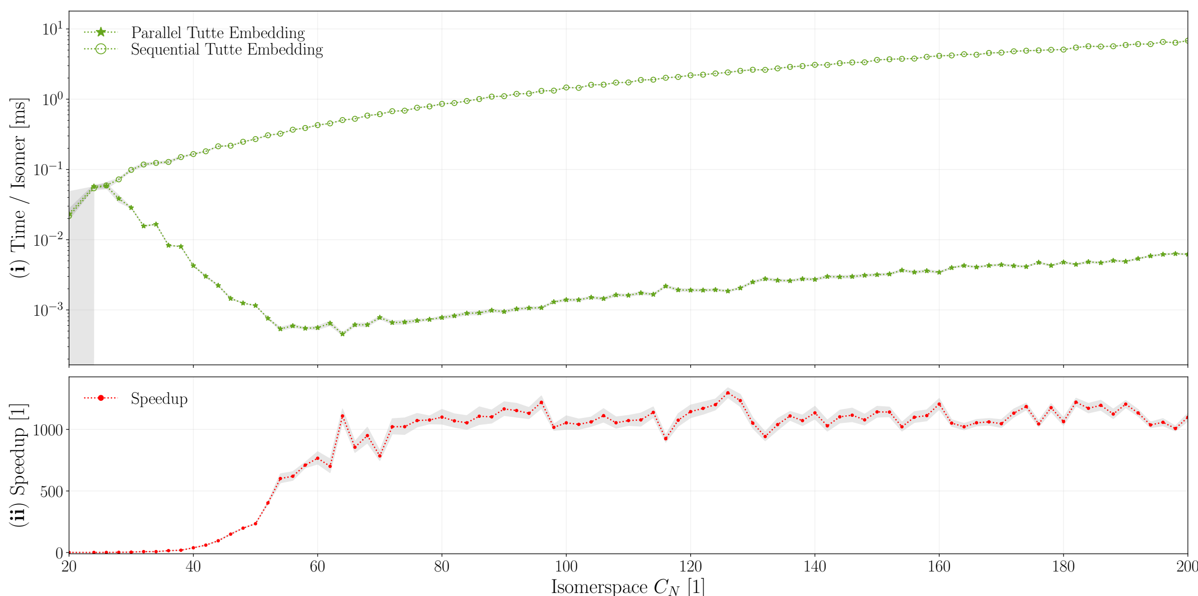


Figure 5.4: ***(j)*** *Runtime per isomer of $\star$ parallel and $\circ$ sequential Tutte embedding methods. The runtime is plotted on a logarithmic scale to allow for both to be visible.* ***(ii)*** *Speedup of parallel kernel relative to sequential method plotted as a function of the isomerspace number C_N. Shaded area corresponds to $\pm 2\sigma$*

As noted in the description of Algorithm 21 we expect the sequential method to scale with $\mathcal{O}(N^2)$ and the parallel version to scale with $\mathcal{O}(log_2(N)N^2)$. To test this we plot the (**k**) runtime, (**ii**) runtime per node (N) and (**ii**) runtime per node squared (N^2) for both the parallel and sequential implementations in Fig. 5.5. And curiously it appears in Fig. 5.5 (**iii**) that both parallel and sequential implementations scale slightly worse than $\mathcal{O}(N^2)$. We reason that since the number of mathematical operations performed within a single iteration of solving the Tutte embedding, definitely increases at a linear rate, it must be the number of iterations required to converge, that increases super-linearly. Ultimately the assumption that our naive iterative solver converges in $\mathcal{O}(N)$ iterations seems to be false, a more robust iterative solver, like conjugated gradient, could be implemented to remedy this if deemed necessary.

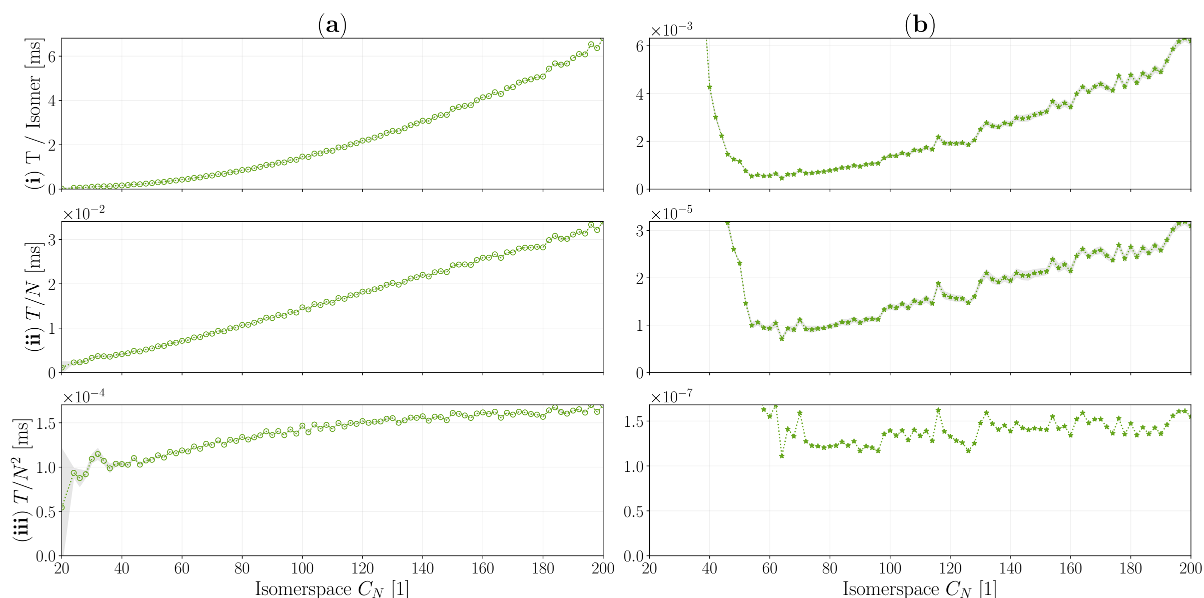


Figure 5.5: ***k*** *Runtime per isomer,* ***ii*** *runtime per node* (N)*,* ***(iii)*** *runtime per node squared* (N^2) *all plotted against isomerspace* C_N*.* ***(a)*** *Sequential CPU Tutte embedding,* ***(b)*** *Parallel lockstep GPU Tutte embedding.*

5.1.6 Pipeline 2: Performance

Now that the Tutte embedding has been parallelised we are able to write an updated pipeline which now consists of the following steps:

1. **BuckyGen**: ○ Generating isomers using the `BuckyGen` generator.
2. **Dualisation**: ○ Dualising the `Graph` object to generate the cubic graph.
3. **Input Conversion**: ● (Overhead) Inserting the `Graph` object into an `IsomerBatch`.
4. **2D Tutte Embedding**: ⋆ (Parallel) Lockstep parallel Tutte embedding of the `IsomerBatch`
5. **Output Conversion**: ● (Overhead) Extracting data from the `IsomerBatch` and inserting `Polyhedron` objects into a `std::queue<Polyhedron>`
6. **Projection**: ○ Sequential spherical projection of the `Polyhedron` objects.
7. **Input Conversion**: ● (Overhead) Inserting `Polyhedron` objects into an `IsomerQueue`.
8. **FF Optimisation**: ⋆ (Parallel) Lockstep parallel queue enabled forcefield optimisation of an `IsomerBatch` (see Fig. 4.13)

This pipeline was implemented and benchmarked in the script at appendix Listing C.3. The steps have been labelled with their corresponding markers from Fig. 5.6. Note how some steps have been compounded into categories: Overhead corresponds to the insertion and extraction of isomers to and from `IsomerBatches`

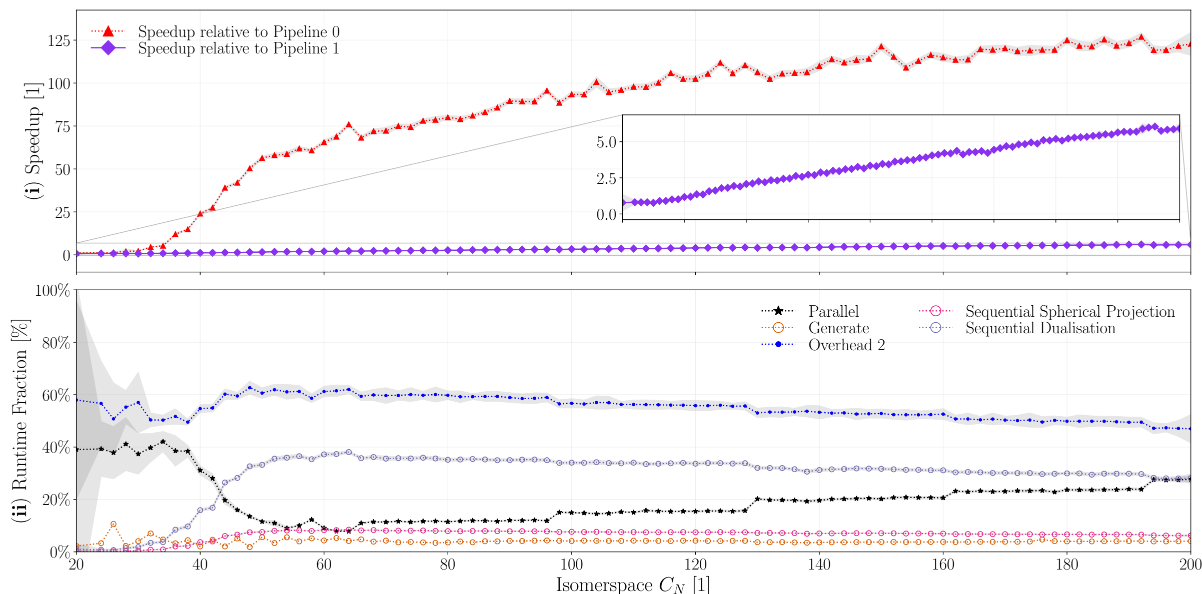


Figure 5.6: ***i*** *Pipeline speedup relative to the baseline sequential pipeline.* ***ii*** *fractional breakdown of runtime contributions from the most impactful remaining components.*

Indeed, parallelising the Tutte embedding has yielded roughly $7.5\times$ speedup compared to pipeline 2 at isomerspace C_{200} and the total speedup now approaches $115\times$ relative to the baseline sequential pipeline.

The overhead in Fig. 5.6 consists of layout conversion from **Graph** object to **GPU Layout** and the reverse from **GPU Layout** to **Polyhedron** objects and once again from **Polyhedron** objects to **GPU Layout**. This overhead is now responsible for approximately 50% of the total runtime at isomerspace C_{200}. However, the majority of the overhead is tied down to the fact that we wish to first compute on the CPU, then on the GPU, then CPU, and finally on the GPU again, if we are able to compute the spherical projection on the GPU as well there will be no need to convert data layout more than once. This is the motivation for parallelising the spherical projection algorithm.

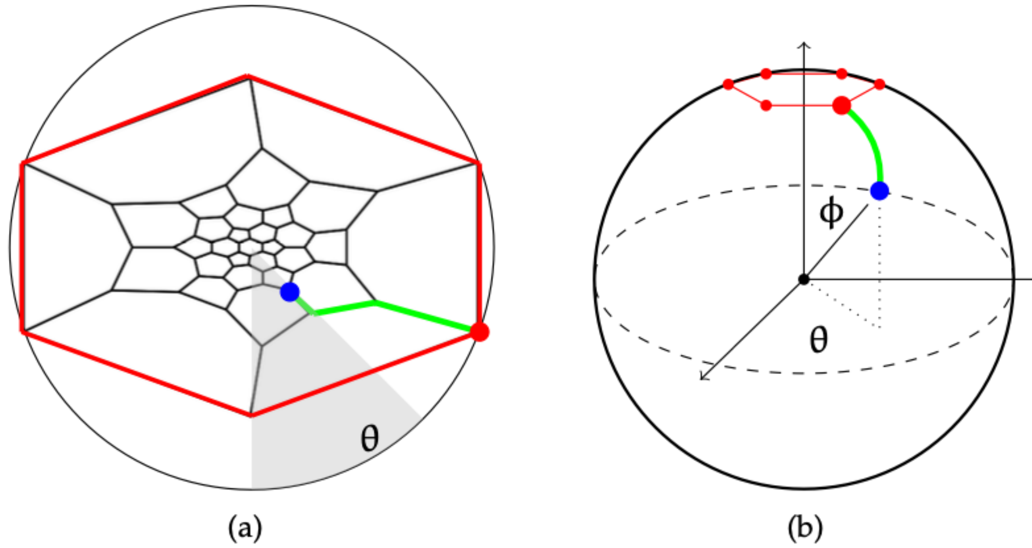


Figure 5.7: *Illustrates how the Tutte embedding (a) is projected onto a sphere. The red outline marks the outer face f_O from which the shortest paths are computed. The green line shows the shortest path from the outer face to the blue node. The position of the blue node with respect to some reference direction gives the azimuthal angle θ and the distance from the outer face gives allows us to compute the polar angle ϕ.*[12]

5.2 Pipeline 4: Spherical Projection

To generate a starting geometry we use a relatively computationally inexpensive method of projecting the 2D embedding found from the Tutte embedding Section 5.1.1, onto a sphere. Let d_v be the topological distance from the outer face to the vertex v and d_{max} be the maximum topological distance from the outer face to any vertex. Then we define the polar angle,

$$\phi_v = \frac{(d_v + 1/2)\pi}{d_{max} + 1} \tag{5.3}$$

And the azimuthal angle,

$$\theta_v = \tan^{-1}((\boldsymbol{X^{2D}}_v - \boldsymbol{C}_v)_x, (\boldsymbol{X^{2D}}_v - \boldsymbol{C}_v)_y) \tag{5.4}$$

Fig. 5.7 visualizes how such a planar Tutte embedding is projected onto a sphere.

Algorithm 22 Sequential Spherical Projection Algorithm

1: **function** SPHERICALPROJECTION($\boldsymbol{G}, \boldsymbol{X^{2D}}, \boldsymbol{X}$) ▷ Graph, 2D embedding
2: $\boldsymbol{C}$ ▷ $(d_{max}+1) \times 2$ centroids for each depth group
3: $\boldsymbol{D} \leftarrow \mathsf{MSSP}(\boldsymbol{f_O}, \boldsymbol{G})$ ▷ $N \times 1$ Topological distances from outer face f_O
4: $\boldsymbol{V}$ ▷ $d_{max}+1$ lists of vertices at each distance
5: $\boldsymbol{\Phi}$ ▷ $N \times 1$ polar angles
6: $\boldsymbol{\Theta}$ ▷ $N \times 1$ azimuthal angles
7: **for** $j \leftarrow 0$ **to** N **do**
8: $\mathsf{append}(\boldsymbol{V}[\boldsymbol{D}[j]], j)$ ▷ Add j^{th} vertex to the list according to topological distance
9: **end for**
10: **for** $d \leftarrow 0$ **to** d_{max} **do**
11: $\boldsymbol{C}[d] \leftarrow (0,0)$ ▷ Centroid at depth d
12: **for** $v \in \boldsymbol{V}[d]$ **do**
13: $\boldsymbol{C}[d] \leftarrow \boldsymbol{C}[d] + \boldsymbol{X^{2D}}[v]/\mathsf{size}(\boldsymbol{V}[d])$ ▷ Compute centroid
14: **end for**
15: **for** $v \in \boldsymbol{V}[d]$ **do**
16: $\boldsymbol{\Phi}[\boldsymbol{v}] \leftarrow (d+1/2) \times \pi/(d_{max}+1)$
17: $r \leftarrow \boldsymbol{X^{2D}}[v] - \boldsymbol{C}[d]$
18: $\boldsymbol{\Theta}[v] \leftarrow \mathsf{atan2}(r_x, r_y)$
19: **end for**
20: **end for**
21: **for** $j \in \{0, \ldots N-1\}$ **do**
22: $x \leftarrow \cos(\boldsymbol{\Theta}[j]) \times \sin(\boldsymbol{\Phi}[j])$ ▷ Conversion from spherical to Cartesian coordinates
23: $y \leftarrow \sin(\boldsymbol{\Theta}[j]) \times \sin(\boldsymbol{\Phi}[j])$
24: $z \leftarrow \cos(\boldsymbol{\Phi}[j])$
25: $\boldsymbol{X}[j] \leftarrow (x, y, z)$
26: **end for**
27: $c_m \leftarrow (0,0,0)$ ▷ Center of mass
28: **for** $j \in \{0, \ldots N-1\}$ **do**
29: $c_m \leftarrow c_m + \boldsymbol{X}[j]/N$
30: **end for**
31: **for** $j \in \{0, \ldots N-1\}$ **do**
32: $\boldsymbol{X}[j] = \boldsymbol{X}[j] - c_m$ ▷ Move centre of mass to the origin $(0,0,0)$
33: **end for**
34: $\bar{r} \leftarrow 0$ ▷ Average distance to neighbouring vertices in the graph
35: **for** $j \in \{0, \ldots N-1\}$ **do**
36: **for** $k \leftarrow 0$ **to** 2 **do**
37: $w \leftarrow \boldsymbol{G}[j,k]$
38: $\bar{r} \leftarrow \bar{r} + \mathsf{norm}(\boldsymbol{X}[j] - \boldsymbol{X}[w])/(3 \times N)$
39: **end for**
40: **end for**
41: **for** $j \in \{0, \ldots N-1\}$ **do**
42: $\boldsymbol{X}[j] = s \times \boldsymbol{X}[j]/\bar{r}$ ▷ Normalize and scale spherical projection
43: **end for**
44: **return** $\boldsymbol{X}$
45: **end function**

In order to compute spherical coordinates we have to first determine the topological distance from the source vertices u in the outer face f_O to every other vertex $v \notin f_O$ we need to compute the *multiple source shortest paths* or MSSPs.

Algorithm 23 Sequential MSSPs Algorithm

```
 1: function MSSPs(F_0, G)                        ▷ Outer face, Cubic graph
 2:     S                                ▷ Source vertices maximum size is N
 3:     G                                     ▷ N × 3 cubic adjacency list
 4:     Q_l                                         ▷ First in first out queue
 5:     D              ▷ N × 1 Topological distance to vertices in the graph
 6:     for j ∈ {0, . . . N − 1} do
 7:         D[j] ← ∞                       ▷ Distances are ∞ before discovery
 8:     end for
 9:     for s ∈ S do
10:         D[s] ← 0                         ▷ The distance to the sources is 0
11:         push(Q_l, s)                            ▷ We start at the sources
12:     end for
13:     while size(Q_l) > 0 do
14:         v ← pop(Q_l)
15:         for k ← 0 to 2 do
16:             w ← G[v, k]
17:             if D[w] = ∞ then             ▷ This node has not been visited before
18:                 D[w] ← D[v] + 1
19:                 push(Q_l, w)
20:             end if
21:         end for
22:     end while
23:     return D                              ▷ Return topological distances
24: end function
```

The MSSPs algorithm is implemented using a *first in first out* or *FIFO* queue using a *breadth-first search* (BFS) approach. We start out by initialising the topological distance array $\boldsymbol{D}$ to ∞, in reality we may choose something like -1 or 2^{31}, as a flag for whether a node has been visited previously. Then set the source distances to 0 and push the source vertices $\boldsymbol{S}$ (in our particular case the sources are the vertices of the outer face f_O) to the queue $\boldsymbol{Q}_l$. Now we enter the main part of the algorithm. First we let v be the first element of the queue, which is popped from the front, and continue to check if its neighbours have been visited before. If a neighbour w has not been seen before we know that it must be topologically 1 further away than v, and we finally add this neighbour w to the queue. This way we traverse the nodes with the shortest topological distance first and since all edges $\boldsymbol{E}$ have a weight of 1 we know that the distances are indeed the shortest possible distances. This process is repeated until all vertices in the graph have been visited. The runtime complexity of the sequential MSSPs algorithm is $\mathcal{O}(N)$ as both push and pop operations on a queue are $\mathcal{O}(1)$ and these operations are performed $\mathcal{O}(N)$ times.

Once we have computed the topological distances from f_O we are able to compute the spherical projection. First we fill $\boldsymbol{V}$ sorted by distance such that the jth list $\boldsymbol{V}_i$ contains vertices that are j distance from f_O. Now we compute the centroids of all vertices at each depth $\boldsymbol{C}_d$, then we compute the azimuthal and polar angles according to equations Eq. (5.4) and Eq. (5.3). Once the spherical coordinates, assuming a unit sphere $r = 1$, have been computed we can trivially convert to Cartesian coordinates. In the final part of the algorithm, lines 29 - 45, we compute the centre of mass c_m by summing up the coordinates, each with same mass, and dividing by the total number of vertices/atoms N. We then subtract c_m from each coordinate in $\boldsymbol{X}$, effectively centering the coordinate system. Finally, we compute the average Cartesian distance between vertices $\bar{r}$ and use this in conjunction with a scale factor s to produce a normalized scaling of the coordinate system. The purpose of this last step is entirely empirical, as it has proven more robust to separate the vertices by a scale factor of $s \approx 4$. Assuming that enough memory can be reserved for $\boldsymbol{V}$ ahead of time, as memory allocations have do not in general have guarantees about runtime complexity, we can model it as a constant that only has to happen once $\mathcal{O}(1)$. The MSSPs embedded in the algorithm has complexity $\mathcal{O}(N)$. Furthermore, since the remaining operations in Algorithm 22 are all simple arithmetic operations or reads/writes from simple arrays and the largest for-loop is N iterations the total runtime complexity must also be $\mathcal{O}(N)$.

5.2.1 Parallel Spherical Projection Algorithm

Unfortunately the otherwise work-efficient linear time complexity MSSPs algorithm implemented sequentially Algorithm 23 is not trivial to parallelise as it makes use of a queue which is not a structure which lends itself very well to concurrency, as the order of insertions and removals from the queue object is paramount to the function of the algorithm. For these reasons the algorithm is unchanged, and we only exploit the isomer-level of parallelism and use a single processing element to compute the MSSPs per isomer. In order to implement Algorithm 23 in CUDA a sequential queue implementation was required, as no such queue is provided in CUDA or CUB by default, a simple cyclical array-based queue was written in CUDA device_deque.cu. The queue takes a pointer to a piece of memory provided by the user and a capacity, it is the users' responsibility to ensure that no pointer aliasing occurs, this design was chosen as dynamic memory allocation is expensive, and we wish to be able to exploit the L1 cache on NVIDIA hardware by passing a pointer to cache.

Algorithm 24 Parallel Spherical Projection Algorithm

1: **function** SPHERICALPROJECTION(G_{in}, X^{2D}_{in}, X_{out}) ▷ Cubic Graphs, 2D Embeddings, 3D Embeddings
2: **for** $i \in \{0, \dots M-1\}$ **do in lockstep** ▷ For each isomer
3: $G \leftarrow G_{in}[i]$ ▷ Input Graph
4: $X^{2D} \leftarrow X^{2D}_{in}[i]$ ▷ Input 2D Embedding
5: $D \leftarrow$ MSSPs(F_0, G) ▷ $N \times 1$ topological distance from f_O to other vertices
6: C ▷ $(d_{max}+1) \times 2$ centroids for each depth group
7: N_d ▷ $(d_{max}+1) \times 1$ number of vertices at each distance
8: Φ ▷ $N \times 1$ polar angles
9: Θ ▷ $N \times 1$ azimuthal angles
10: R ▷ $N \times 1$ mean neighbour distances
11: **for** $j \leftarrow 0$ **to** d_{max} **do in lockstep** ▷ For each atom
12: $N_d[j] \leftarrow 0$ ▷ Initialise number of vertices at each depth
13: **end for**
14: **for** $j \leftarrow 0$ **to** $N-1$ **do in lockstep** ▷ For each atom
15: atomic_add$(N_d[D[j]], 1)$ ▷ Compute number of vertices at each depth
16: **end for**
17: **for** $j \leftarrow 0$ **to** $N-1$ **do in lockstep** ▷ For each atom
18: $d \leftarrow D[j]$ ▷ Get depth of vertex
19: atomic_add$(C[\mathrm{d}], X^{2D}[\mathrm{j}]/N_d[\mathrm{d}])$ ▷ Compute centroid at depth d
20: **end for**
21: **for** $j \leftarrow 0$ **to** $N-1$ **do in lockstep** ▷ For each atom
22: $\Phi[j] \leftarrow (D[j]+1/2) \times \pi/(d_{max}+1)$ ▷ Compute polar angle using Eq. (5.3)
23: $r \leftarrow X^{2D}[j] - C[D[j]]$ ▷ Compute radial vector
24: $\Theta[j] \leftarrow$ atan2(r_x, r_y) ▷ Compute azimuthal angle using Eq. (5.4)
25: **end for**
26: **for** $j \in \{0, \dots N-1\}$ **do in lockstep** ▷ For each atom
27: $x \leftarrow \cos(\Theta[j]) \times \sin(\Phi[j])$ ▷ Conversion to Cartesian coordinates
28: $y \leftarrow \sin(\Theta[j]) \times \sin(\Phi[j])$
29: $z \leftarrow \cos(\Phi[j])$
30: $X[j] \leftarrow (x, y, z)$ ▷ Store new coordinates
31: **end for**
32: $c_m \leftarrow$ reduce$(X, +)/N$ ▷ Center of mass computed using a reduction
33: **for** $j \in \{0, \dots N-1\}$ **do in lockstep** ▷ For each atom
34: $X[j] = X[j] - c_m$ ▷ Move centre of mass to the origin $(0,0,0)$
35: **end for**
36: **for** $j \in \{0, \dots N-1\}$ **do in lockstep** ▷ For each atom
37: $R[j] \leftarrow 0$ ▷ Initialise distance array
38: **for** $k \leftarrow 0$ **to** 2 **do**
39: $w \leftarrow G[j, k]$
40: $R[j] \leftarrow R[j] + \mathrm{norm}(X[j] - X[w])/(3 \times N)$
41: **end for**
42: **end for**
43: $\bar{r} \leftarrow$ reduce$(R, +)$ ▷ Compute mean distance
44: **for** $j \in \{0, \dots N-1\}$ **do in lockstep**
45: $X[j] = s(X[j]/\bar{r})$ ▷ Normalize and scale spherical projection
46: **end for**
47: $X[i] \leftarrow X$ ▷ Store result
48: **end for**
49: **end function**

Algorithm 22 presents further challenges in terms of parallelism in lines 12-19, because the lists $V_0, V_1 \cdots V_{d_max}$ have unknown length and would require some form of dynamic parallelism to properly parallelise, and while this is in general fine on CPUs, GPUs and other wide vector-hardware are notoriously ill-suited for this. As the general level of degree of parallelism is N we elect to stick to this level of parallelism and modify the problem slightly to fit in the SIMT-paradigm. Instead of sequentially creating and sorting the list of vertices $\boldsymbol{V}$ as in Algorithm 22, we notice that each vertex has a depth associated with it $\boldsymbol{D}[j]$, now each PE simply has to add its coordinate $\boldsymbol{X}[j]$ to the centroid it belongs to $\boldsymbol{C}[\boldsymbol{D}[j]]$. We note that since multiple processing elements will access the same element in $\boldsymbol{C}$ we must take care to avoid *race conditions*. While this could be resolved with reductions they would be of unknown size and inevitably require dynamic parallelism, the very problem we are trying to solve, therefore we turn to *atomic* additions. Technically this avoids race conditions and theoretically this yields the correct answer since addition is associative, however floating-point addition is not associative and thus the results are non-deterministic. Whether this is acceptable depends on the use case, however for the purposes of reproducibility of this thesis and the results within, we have chosen to deal with this using a specific work-order of atomics through synchronisation primitives. This leads to an unknown runtime complexity for the MSSPs algorithm, since atomics are implementation specific. The conversion from spherical coordinates to Cartesian coordinates is entirely data parallel and thus $\mathcal{O}(N)$. Finally, the coordinate transformation and scaling of operations are dealt with using simple N element wide reductions, the parallel runtime complexity of the reduction operation is $\mathcal{O}(N \log_2(N))$. Thus, the total runtime complexity is presumably bounded by the MSSPs. Further work may be done to rethink the MSSPs algorithm for massively parallel hardware and parallelisability of the centroid computations. There is however, as of writing, no cause for concern if this algorithm is used in conjunction with forcefield optimisation as forcefield optimisation scales with $\mathcal{O}(N^2)$ and performs a much greater number of operations per atom.

5.2.2 Validity of Parallel Spherical Projection Algorithm

As with all other floating-point based algorithms, due to the non-associative nature of operations, some reasonable discrepancy in the results must be tolerated. We use the same 1000 isomers randomly sampled as discussed in the Tutte-embedding validationSection 5.1.4. For the validity analysis of the spherical projection we perform first an accuracy assessment using Tutte embeddings generated using the sequential Algorithm 20 followed by spherical projection on the GPU, and a second analysis starting from Tutte embeddings generated in parallel using Algorithm 21. We do this to check the impact on initial geometry of cumulative rounding error.

It becomes clear from Fig. 5.8, the double pronged shape in the absolute density map propagates from the Tutte-embedding error profile Fig. 5.3. Furthermore, we see that indeed some increase in relative error can be observed when going from spherical projection performed on sequentially embeddings (**a**), to projection performed on GPU computed embeddings (**b**). The relative errors are still very reasonable and whether

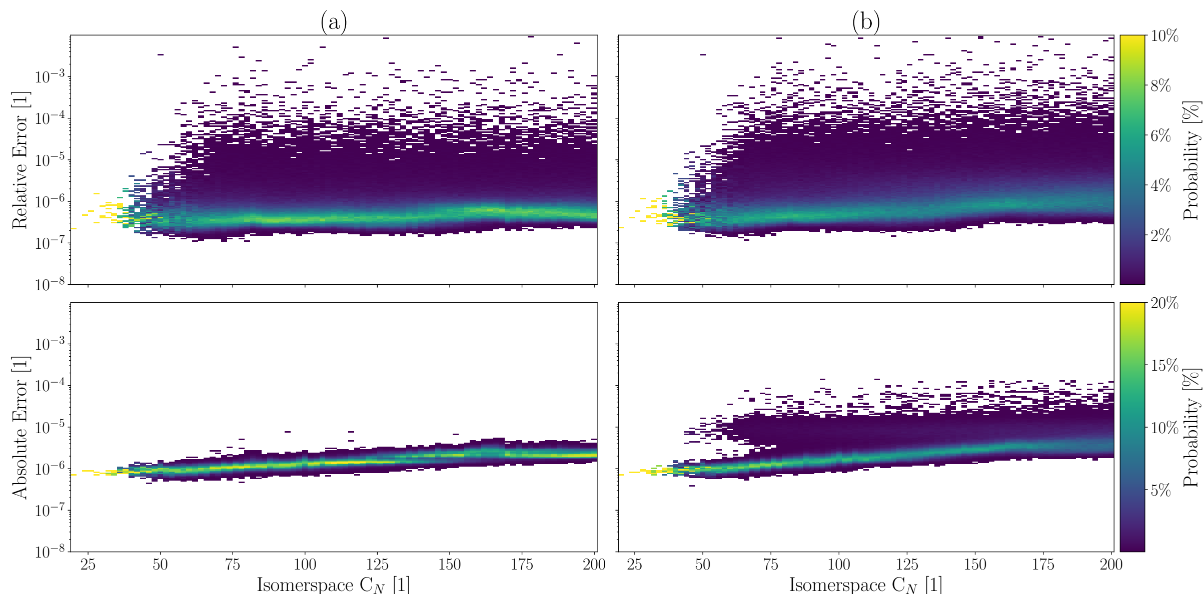


Figure 5.8: *Relative error probability density maps (**top**) and absolute error Probability density maps (**bottom**). In column (a) The Tutte-embedding is performed on the CPU and in column (b) the Tutte-embedding is performed on the GPU.*

this increased discrepancy from sequential generated geometry, actually results in worse quality is not clear.

5.2.3 Performance of Parallel Spherical Projection Algorithm

The performance of the parallel spherical projection algorithm is measured in the same script from Listing C.9 from Fig. 5.9 we see that performance peaks at speedup of ≈ 210 around isomerspace C_{62} which is sensible as this is just beyond the saturation point, beyond this parallelism only decreases for the reasons discussed in the description of Algorithm 24. Indeed, performance drops below 100-fold speedup already at isomerspace C_{194}. While the fully parallelised components of Algorithm 24 may initially have dominated the runtime cost we certainly see the impact of the sequential MSSPs as $N \rightarrow 200$.

As with the Tutte embedding, we similarly analyse the scaling behaviour of the spherical projection algorithm by plotting runtime per isomer T/M, node $T/(M * N)$ and node squared $T/(MN^2)$ (Fig. 5.10). The runtime per node of the sequential algorithm (Fig. 5.10 **ii**,**a**) is constant, verifying that runtime complexity is indeed $\mathcal{O}(N)$ as expected. The parallel algorithm on the other hand has much worse scaling as expected, more detailed performance analysis is required to verify the predicted $\mathcal{O}(N^2)$ parallel runtime complexity as the $\mathcal{O}(log_2(N)N)$ are seemingly still significant at isomerspace C_{200}.

5.2.4 Pipeline 4: Performance

While the spherical projection algorithm undoubtedly has worse scaling characteristics than we would like, the projection is itself a very small fraction of the total computational cost, so it turns out to be rather insignificant. Furthermore, the real purpose

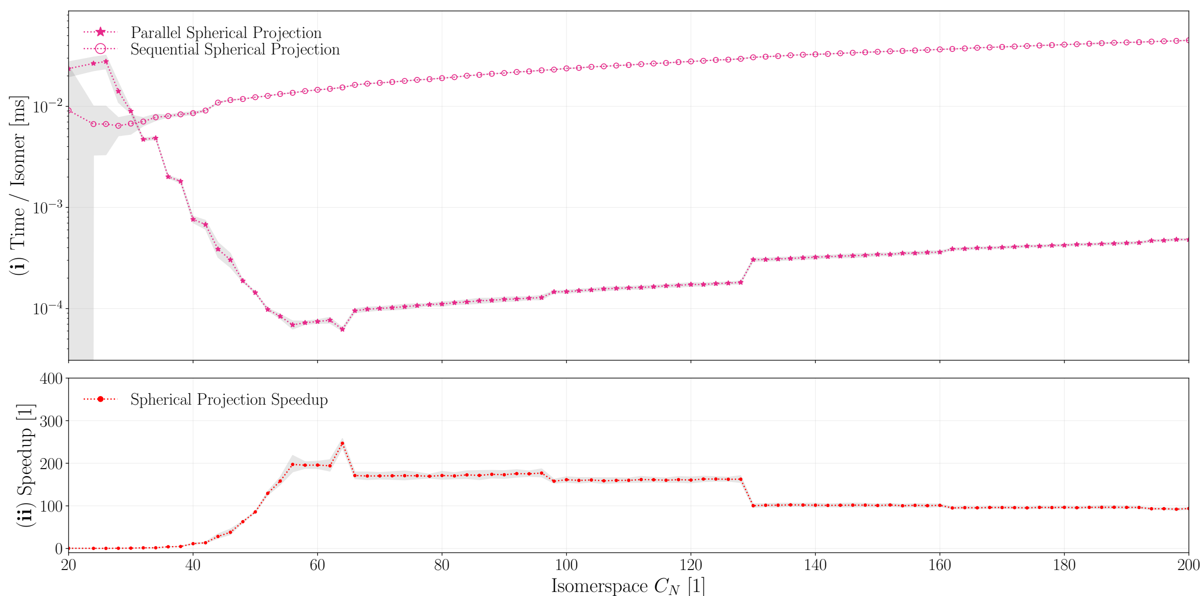


Figure 5.9: ***(j)*** *Runtime per isomer of ⋆ parallel and ∘ sequential spherical projection methods. The runtime is plotted on a logarithmic scale to allow for both to be visible.* ***(ii)*** *Speedup of parallel kernel relative to sequential method plotted as a function of the isomerspace number C_N. Shaded area corresponds to $2 \pm \sigma$.*

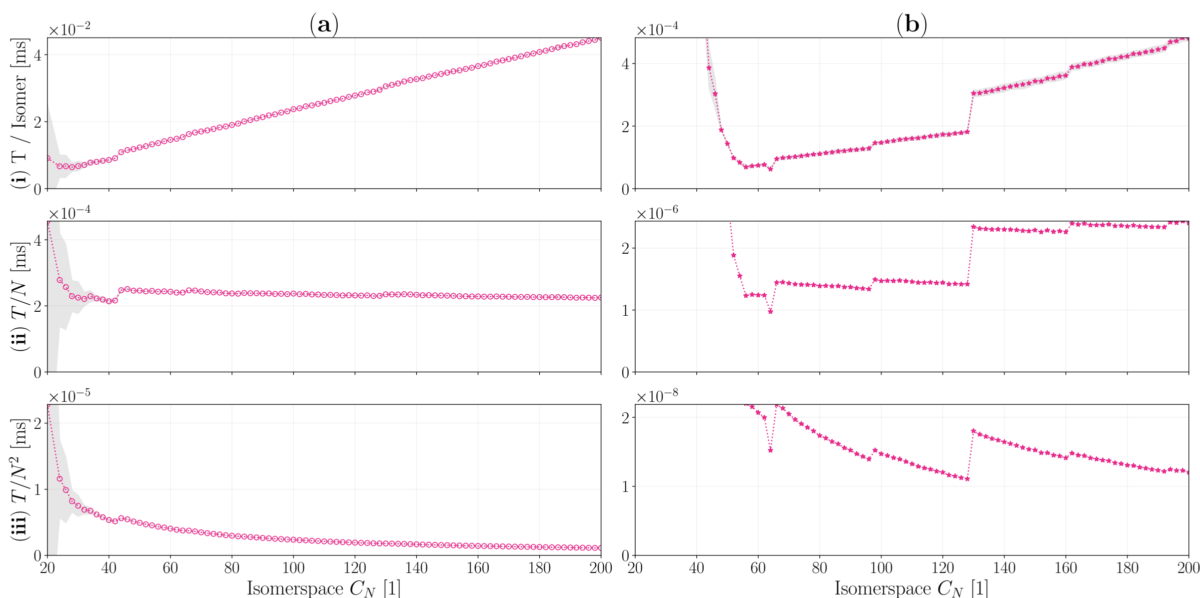


Figure 5.10: ***j*** *Runtime per isomer,* ***ii*** *runtime per node (N),* ***(iii)*** *runtime per node squared (N^2) all plotted against isomerspace C_N.* ***(a)*** *Sequential CPU spherical projection,* ***(b)*** *parallel lockstep GPU spherical projection. Shaded areas represent $\pm 2\sigma$*

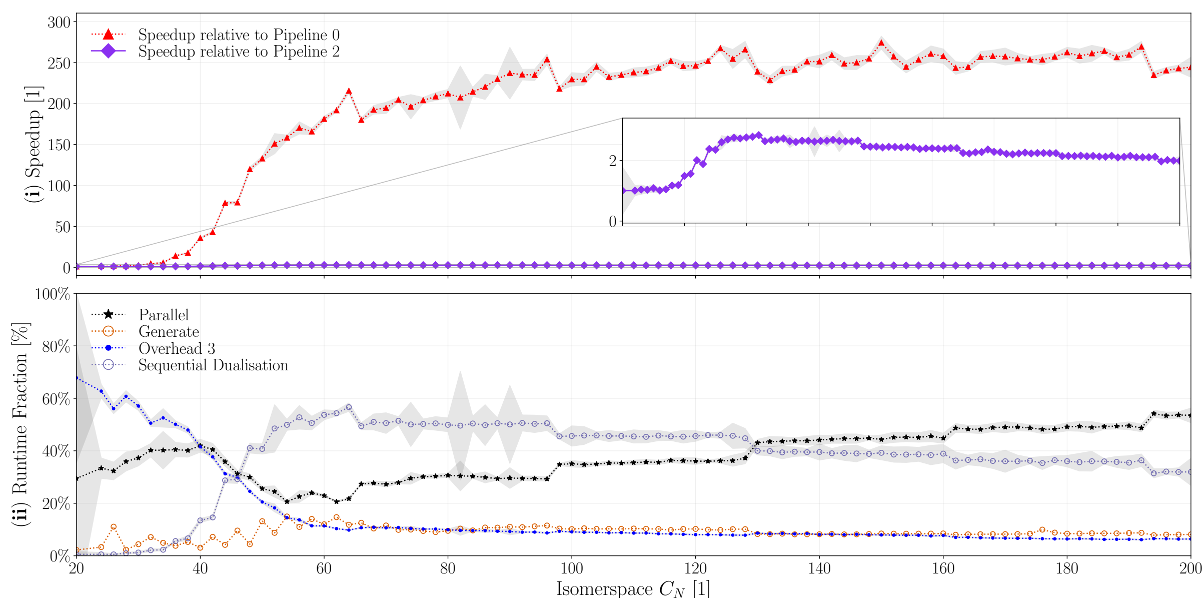


Figure 5.11: *(j) The speedup of pipeline 4 relative to pipeline 3 (♦) and speedup relative to the sequential baseline ▲. (ii) Breakdown of fractional runtime comprised of, parallel components, dualisation, BuckyGen generation and overhead. Shaded areas represent* $\pm 2\sigma$

of implementing a parallel version was to move computation and thus data transfer and data layout conversion away from the CPU. We explore now a new more parallel pipeline comprised of the following components:

1. **BuckyGen**: ○ (Sequential) Generating isomers using the `BuckyGen` generator.
2. **Dualisation**: ○ (Sequential) Dualising the resulting `Graph` objects.
3. **Input Conversion**: ● (Overhead) Inserting the `Graph` object into an `IsomerBatch`.
4. **2D Tutte Embedding**: ⋆ (Parallel) Lockstep parallel Tutte embedding of the `IsomerBatch`.
5. **Projection**: ⋆ (Parallel) Semi parallel spherical projection of the `IsomerBatch`.
6. **Insertion** ● (Overhead) Inserting the `IsomerBatch` into an `IsomerQueue`.
7. **FF Optimisation**: ⋆ (Parallel) Lockstep parallel queue enabled forcefield optimisation of an `IsomerBatch`.

Pipeline 4 has been benchmarked using the script in Listing C.4.

Naturally we see that as more of the program becomes parallel and overhead is reduced the performance improvements are increasingly modest. We find that removing data layout conversions and construction of polyhedron objects has increased performance by a factor of $2-3\times$ in the isomerspace range from C_{60} to C_{200} and a total pipeline

speedup compared to baseline of $220\times$ for C_{200}. Furthermore, note that the parallel runtime fraction has increased to about $44\% \pm 2\%$ for isomerspace C_{200}. In Fig. 5.11 (**ii**) we see that the most impactful remaining sequential component is the dualisation algorithm, constituting between 70% and 44% of the total runtime in the range C_{50} to C_{200}. For this reason we investigate how to parallelise the dualisation algorithm.

5.3 Pipeline 5: Dualisation

The dual graph $\boldsymbol{G}^*$ of a planar graph $\boldsymbol{G}$ is a graph that has a vertex for each face in $\boldsymbol{G}$ and an edge for every pair of neighbouring faces in $\boldsymbol{G}$. Dual graphs are not unique in general but for three-connected graphs they are, therefore $\boldsymbol{G}^*$ represents the same information as $\boldsymbol{G}$ and the dual operation is its own inverse $(G^*)^* = G$. Therefore, dual graph could refer to either of two representations, we shall refer to the representation where each atom is a vertex as the cubic graph $\boldsymbol{G}$ and the dual of this graph as the *dual* $\boldsymbol{G}^*$.

The `BuckyGen`[5] program produces triangulated dual graphs in a canonical clockwise adjacency list representation. That is graphs are represented by adjacency list $\boldsymbol{G}^*$ with $\boldsymbol{G}^*[i,j]$ being the jth neighbouring vertex of the ith vertex. Specifically a planar graph is said to be a plane triangulation if the boundary of each face contains exactly three edges. The ordering of neighbouring vertices in $\boldsymbol{G}^*$ is what affords a particularly trivial way to identify and number the faces (triangles)in the dual graph. Many algorithms revolve around the atoms of the fullerene rather than the pentagons and hexagons of the fullerenes thus an algorithm for computing the cubic graph $\boldsymbol{G}$ from $\boldsymbol{G}^*$ is required. Specifically we wish to produce the adjacency list $\boldsymbol{G}$ which has dimensions $N \times 3$ from the adjacency list $\boldsymbol{G}^*$ with dimension $N_f \times 6$.

5.3.1 Sequential Algorithm

The first step of the algorithm is to iterate over the vertices u and add all triangles to an array $\boldsymbol{T}$ that have not yet been added. However since each triangle is made up of 3 vertices it follows that each triangle may be found 3 times, to solve we use a hashmap $\boldsymbol{D}$ intended to store all arcs N_{de} and their discovery status. Now we can check if an arc has been seen before, if it has not been seen we proceed to append the triangle corresponding to that arc and assign $true$ to all arcs comprised by the triangle.

The second step of the algorithm is to assign numbers to each of the triangles in $\boldsymbol{T}$ these triangles already have the trivial ordering according to their indices $i = [0, N]$ now we wish to be able to identify these numberings based on the vertices the triangle is made up of. We insert the triangles in a hashmap through a hash combination of its constituent vertices. Since every triangle can be represented in 6 different ways we choose the canonical triangle to be the vertices sorted in ascending fashion.

Now we are able to iterate over the triangles in $t \in T$ and the arcs in t. We can identify the jth neighbouring triangle as the one made up of $t_j \to t[\mathsf{mod}(j+1,3)] \to \mathsf{prev}(t_j, t[\mathsf{mod}(j+1,3)])$ now we look up the triangle number assigned to this triangle and assign it to the $(i,j)^{th}$ element in the cubic adjacency list $\boldsymbol{G}$. That is $\boldsymbol{G}[i,j] = \mathsf{find}(T^l, t_n)$, this then concludes the algorithm.

Algorithm 25 Sequential Dualisation Algorithm

```
1: G                                          ▷ Cubic Adjacency List
2: G*                                         ▷ Dual Adjacency List
3: D                 ▷ Hash-map of (arcs, booleans) as (key, value) pair
4: T                                          ▷ Array of triangles
5: T^l                     ▷ Hash-map with of type triangles, integers
6: for u ∈ {0,... N_f − 1} do              ▷ N_f is the number of faces
7:     for i ∈ {0,... Deg_u − 1} do   ▷ Deg_u is the degree of the u^th face
8:         v ← G[u, i]
9:         w ← next_on_face(u, v)
10:        if not find(D, (u, i)) then
11:            T.append((u, v, w))
12:            D[(u, v)] ← true   ▷ Set D to true for all arcs involved in the triangle
13:            D[(v, w)] ← true
14:            D[(w, u)] ← true
15:        end if
16:    end for
17: end for
18: for i ∈ {0,... N − 1} do
19:     t ← sort(T_i)   ▷ Sort the triangle vertices to get well-defined representation
20:     T^l[t] ← i                ▷ Insert the i^th triangle with numbering i
21:     i ← i + 1
22: end for
23: for i ∈ {0,... N − 1} do
24:     t ← T_i
25:     for j ∈ {0, 1, 2} do
26:         u ← t[j]
27:         v ← t[mod(j + 1, 3)]
28:         w ← prev(u, v)        ▷ Vertices u, v, w of the j^th neighbouring triangle
29:         t_n ← sort((u, v, w))
30:         G[i, j] ← find(T^l, t_n)  ▷ Number associated with the neighbouring triangle
31:     end for
32: end for
33: return G
```

Algorithm 26 Parallel Dualisation Algorithm

```
1: function DUALIZE(G*_in, G_out)                          ▷ Dual graphs, Cubic graphs
2:   for i ← to M − 1 do in lockstep                        ▷ For each isomer
3:     O                                                     ▷ N_f × 1 Offset Array
4:     T                                                     ▷ N_f × 6 Sparse arc to triangle ID matrix
5:     R            ▷ N × 2 Representative Arcs for each of the triangles in the dual
6:     G* ← G*_in[i]                                         ▷ N_f × 6 Dual adjacency list
7:     A  ▷ N_f × 6 list of arcs that are representative for a triangle, default value is ∞
8:     for u ∈ {0, . . . N_f − 1} do in lockstep            ▷ For each face
9:       n_a ← 0                      ▷ Number of triangles that the u^th face represents
10:      for j ∈ {0, . . . Deg_u − 1} do             ▷ For each neighbour of the u^th face
11:        (b, c) ← CannonArc(G*, u, G*[u, j])
12:        if b = u then               ▷ If the canonical arc starts with u, then save it
13:          A[u, j] ← c                                     ▷ Save the canonical arc
14:          n_a ← n_a + 1                       ▷ Increment the number of triangles
15:        end if
16:      end for
17:      O[u] = n_a                                  ▷ Store n_a in the offset array
18:    end for
19:    exclusive_scan(O, +)                 ▷ Prefix sum to compute ordering of arcs
20:    for u ∈ {0, . . . N_f − 1} do in lockstep            ▷ For each face
21:      n_a ← 0
22:      for j ∈ {0, . . . Deg_u − 1} do
23:        if A[u, j] ≠ ∞ then
24:          T[u, j] ← O[u] + n_a  ▷ Triangle ID corresponding to {u → A[u, j]}
25:          n_a ← n_a + 1 ▷ Increment #of triangles that the u^th face represents
26:          k = T[u, j]                              ▷ Store the k^th triangle ID
27:          R[k] = (u, A[u, j])     ▷ Store the representative arc for the k^th triangle
28:        end if
29:      end for
30:    end for
31:    for i ∈ {0, . . . N − 1} do in lockstep              ▷ for each triangle
32:      (u, v) ← R[i]                     ▷ Representative arc for the i^th triangle
33:      w ← next(u, v)                          ▷ Third vertex of the i^th triangle
34:      t ← (v, u, w)                      ▷ Vertices comprising the i^th triangle
35:      for j ∈ {0, 1, 2} do           ▷ For each neighbour of the i^th triangle
36:        Representative arc for j^th neighbour
37:        (b, c) ← CannonArc(G*, t[j], t[mod(j + 1, 3)])
38:        k = ArcIdx(G*, b, c)        ▷ Index of the c in the adjacency list of b
39:        G[i, j] ← T[b, k]             ▷ Store triangle ID of the j^th neighbour
40:      end for
41:    end for
42:    G_out[i] ← G
43:  end for
44: end function
```

5.3.2 Parallel Algorithm

There are a number of issues with parallelising the dualisation Algorithm 25, notably hash-maps are notoriously difficult to implement concurrently and unless they are absolutely essential we should refrain from using them. Therefore, we create two lookup tables, $\boldsymbol{T}$ of dimension $N_f \times 6$ containing triangle numbers associated with the $(u,j)^{th}$ arc in $\boldsymbol{G}$, and $\boldsymbol{R}$ of dimension $N \times 2$ a list of all the arcs which represent a triangle. The table $\boldsymbol{T}$ serves a similar purpose to the triangle hashmap in Algorithm 25, with the crucial difference, that we cannot use triangles as keys to access the table. Instead, we must convert the representative arc (u,v) to (u,j) where $\boldsymbol{G}[u,j] = v$, the function $\mathsf{ArcIdx}()$ finds this index j given an arc (u,v) by iterating over the neighbours of u. To fill out these lookup tables in parallel we notice that we can uniquely identify all triangles in parallel by representing each triangle with the arc $t_j \to t[\mathsf{mod}(j+1,3)]$ with the source node $t[j] = min(t)$ being the smallest node in the triangle. Choosing this representation means that each PE_u can now independently ascertain whether they are the representative node by checking that $\mathsf{min}(t) = u$. As such the parallel dualisation algorithm works by having each PE_u iterate over $\forall j \in \{0, Deg_u - 1\}\boldsymbol{G^*}[u,j]$, then for each neighbour $v = \boldsymbol{G}[u,j]$ check whether the minimum node in the triangle $(u, v, \mathsf{next}(u,v))$ is u. If this is the case we store the vertex v in the $\boldsymbol{A}[u,j]$, the arc is implicitly $u \to \boldsymbol{A}[u,j]$, so we only need to store the destination node not the source. We can then keep track of the number of triangles represented by each node n_a such that a unique identifier can be associated with each triangle. We assign this to an offset array $\boldsymbol{O}[u] = n_a$, now we can realize unique identifiers by first performing an *exclusive prefix sum* on the array yielding a list of ascending offsets. This exclusive prefix sum is a subset of the parallel primitive scan, which can be computed efficiently in parallel. With these offsets computed we can finally find the triangle numbers $\boldsymbol{T}$ by having each PE_u iterate over the neighbours of the uth face, if $a_c[j]$ is not ∞ we know that it is a representative arc, and we assign it: $\boldsymbol{T}[u,j] = \boldsymbol{O}[u] + n_a$ and increment n_a. Additionally, we can fill out the $k = \boldsymbol{T}[u,j]^{th}$ element of $\boldsymbol{R}[k] = (u, \boldsymbol{A}[u,j])$ since we now know the triangle number and representative arc.

Finally, having filled out the lookup tables, we proceed to have each PE_u find a triangle $t = (v,u,w)$ where $(u,v) = \boldsymbol{R_u}$ and $w = \mathsf{next}(u,v)$. From this triangle we can find its neighbouring triangles by considering the arcs $v \to u$, $u \to w$ and $w \to v$, from these edges we find the canonical arc (b,c) that represents each of the neighbouring triangles. Iterating over the neighbour arcs $j = 0,1,2$ we can now fill out the cubic adjacency list $\boldsymbol{G}[u,j] = \boldsymbol{T}[b,k]$ where $k = \mathsf{ArcIdx}(b,c)$ and $(b,c) = \mathsf{CannonArc}(\boldsymbol{G}, t[j], t[\mathsf{mod}(j+1,3)])$ is the canonical arc.

All the array accesses and integer arithmetic involved in Algorithm 26 have constant runtime complexity, and they are performed at most order N times therefore these operations have the same runtime complexity ($\mathcal{O}(N)$). The prefix sum however has slightly worse scaling characteristics at $\mathcal{O}(N\log_2(N))$. Thus, the parallel dualisation algorithm has a total runtime complexity of $\mathcal{O}(N\log_2(N))$.

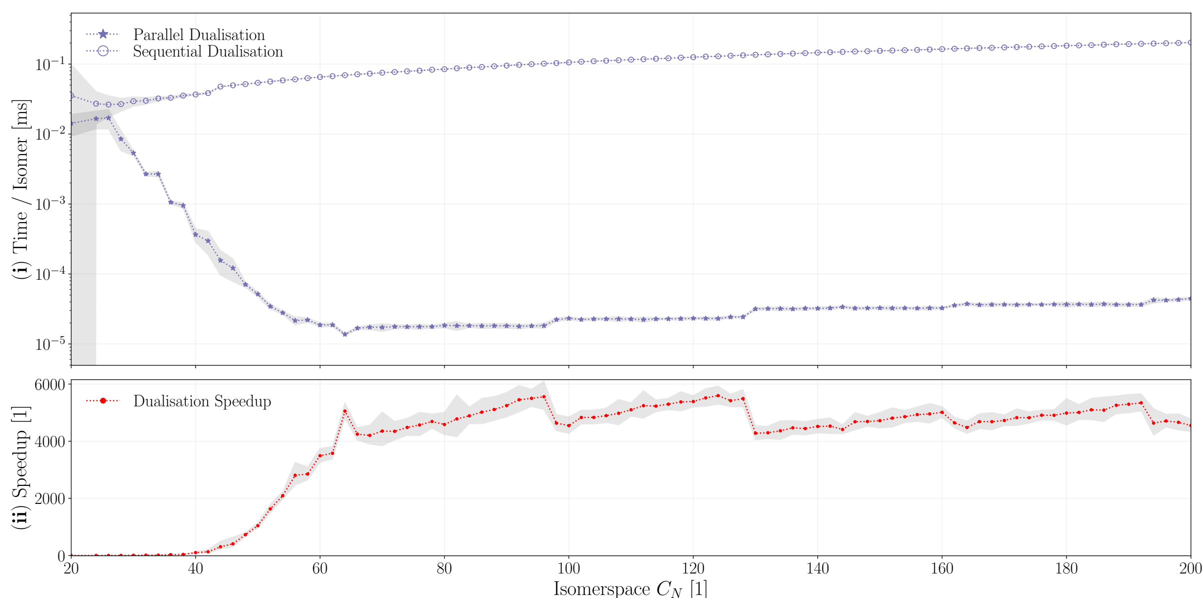


Figure 5.12: *(i) Time per isomer performance for the sequential (○) dualisation algorithm and the parallel lockstep implementation (⋆). (ii) Speedup of parallel kernel relative to sequential method plotted as a function of the isomerspace number C_N. Shaded areas represent $\pm 2\sigma$.*

5.3.3 Validation of Dualisation Algorithm

Validation of the dualisation algorithm trivial compared to evaluating the validity of the floating-point based algorithms, Tutte-embedding Algorithm 21 and spherical projection Algorithm 24. Since dualisation is an exact algorithm consisting only of associative integer maths operations we can just check that every single element in $\boldsymbol{G}$ is identical for the parallel algorithm and the sequential algorithm. We generate the first 10000 isomers in the isomerspaces $C_{20}, C_{24}, \cdots, C_{200}$ and confirm that indeed 100% of the isomers pass the test.

5.3.4 Performance of Dualisation Algorithm

The sequential algorithm is written in C++ using standard library implementations of `unordered_map`. We benchmark the performance of the dualisation algorithm across the isomerspace range $C_{20}, C_{24}, \cdots, C_{200}$ using the script in Listing C.9. The absolute and comparative performance can be seen in figure Fig. 5.12.

We see that, opting for lookup tables and efficient parallel exclusive scan over a potential concurrent hash-map has yielded an exceedingly efficient GPU based dualisation algorithm, reaching between $4000\times$ and $6000\times$ speedup in the isomerspace range $[C_{62}, C_{200}]$ compared with the sequential C++ implementation Algorithm 25. What was previously computed for C_{200} in $200\mu s$ per isomer can now be done in $40ns$ per isomer. As per the discussion about Algorithm 26 we expect $\mathcal{O}(N\log_2(N))$ runtime complexity. However, the base 2 logarithm grows very slowly in our range of interest, $\lceil log_2(20)\rceil = 5$, $\lceil \log_2(200)\rceil = 8$ and the maximum number of blocks per multiprocessor, 16 for architecture 8.6, using `-maxrregcount=40` means that saturation is not reached until blocksize

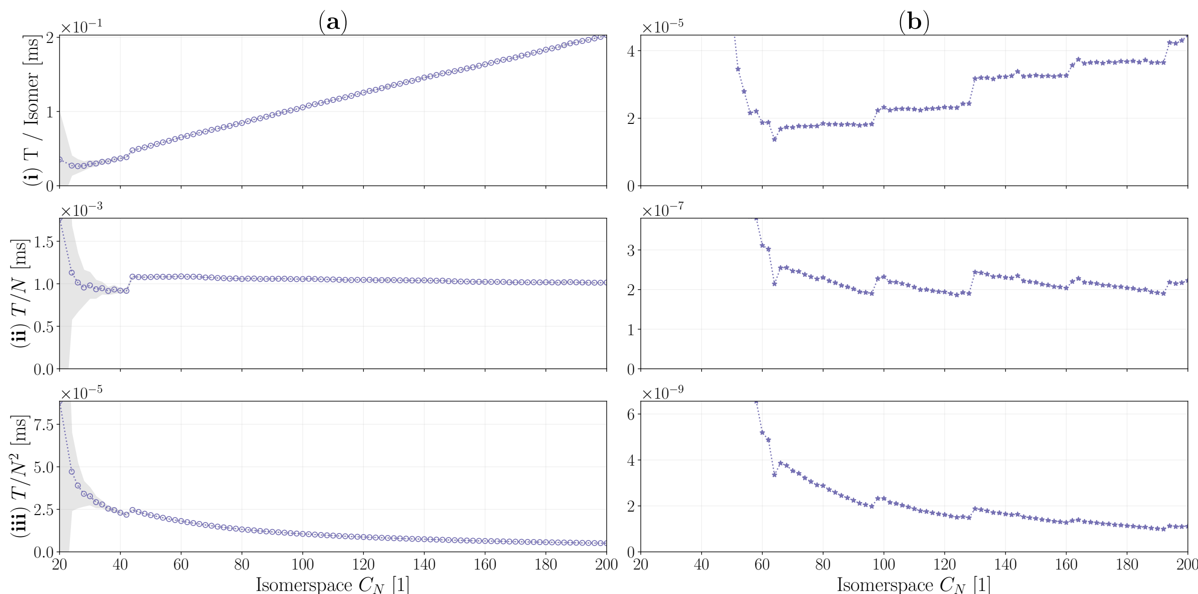


Figure 5.13: *(**i**) Runtime per isomer, (**ii**) runtime per node (N), (**iii**) runtime per node squared (N^2) all plotted against isomerspace C_N. (**a**) Sequential Dualisation, (**b**) Parallel lockstep Dualisation.*

64. Furthermore, the prefix sum is so efficient that it does not yet constitute a large fraction of the dualisation runtime. For these reasons we may not be able to see the $\log_2$ factor in the isomerspace range $[C_{20}, C_{200}]$.

We inspect the runtime complexity of the sequential and parallel implementations in Fig. 5.13, and see that as expected the sequential dualisation algorithm appears to have linear runtime complexity and the parallel dualisation algorithm, while more complex due to discrete hardware resource allocation, still appears to scale almost linearly.

5.3.5 Pipeline 5: Performance

With all components except for the `BuckyGen` generator now running in parallel lockstep on the GPU we construct a new pipeline consisting of the following steps:

1. **BuckyGen**: ○ (Sequential) Generating isomers using the `BuckyGen` generator.
2. **Input Conversion**: ● (Overhead) Inserting the `Graph` object into an `IsomerBatch`.
3. **Dualisation**: ⋆ (Parallel) Dualising an entire `IsomerBatch` in lockstep producing.
4. **2D Tutte Embedding**: ⋆ (Parallel) Lockstep parallel Tutte embedding of the `IsomerBatch`.
5. **Projection**: ⋆ (Parallel) Semi parallel spherical projection of the `IsomerBatch`.
6. **Insertion**: ● (Overhead) Inserting the `IsomerBatch` into an `IsomerQueue`.
7. **FF Optimisation**: ⋆ (Parallel) Lockstep parallel queue enabled forcefield optimisation of an `IsomerBatch`.

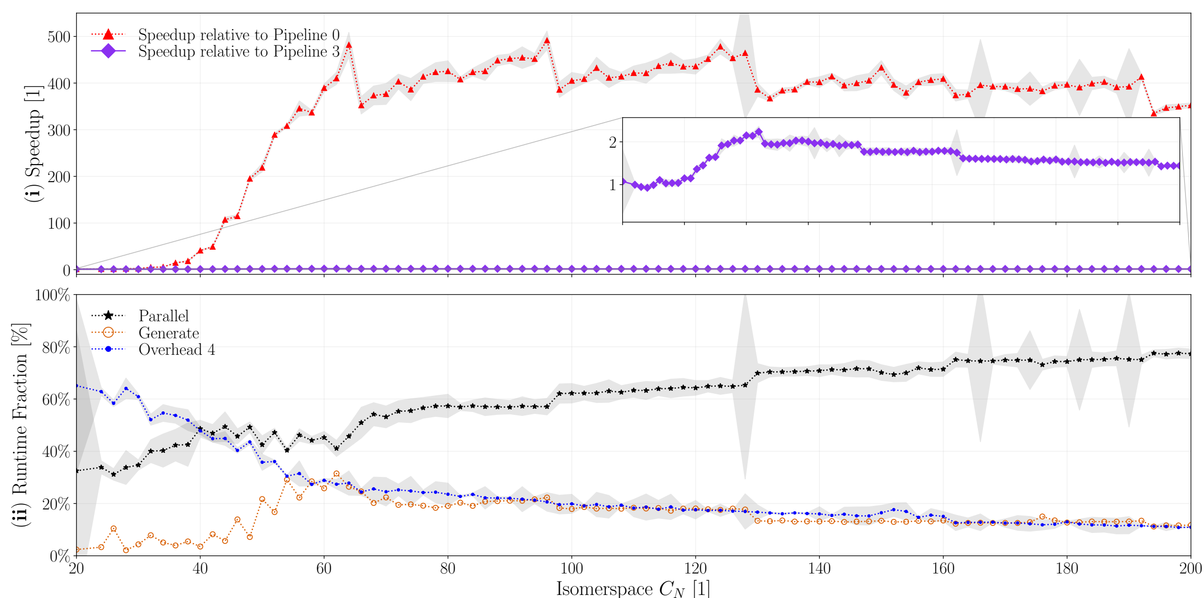


Figure 5.14: *(i) Speedup of pipeline 5 relative to the previous pipeline iteration, pipeline 4 and relative to the baseline pipeline 0. **ii** Fractional breakdown of the remaining runtime components, parallel ($\star$), generate and overhead 4. Shaded areas represent* $\pm 2\sigma$*.*

As with the previous pipelines we benchmark this pipeline using a randomly selected sample of the randomly sampled isomers from isomerspaces $[C_{20}, C_{200}]$, the script can be found in Listing C.5.

In pipeline 5 the overhead only refers to the one necessary data layout conversion from CPU to GPU after generating. We see that pipeline 5 is indeed $3\times - 1.8\times$ faster than pipeline 4 in the range $[C_{50}, C_{200}]$ this is exactly what we expect from trivialising what previously constituted $44(2)\%$ of the runtime (at C_{200}). Finally, pipeline 5, compared to the sequential baseline, is now $400\times$ faster, and we see that overhead and generation of isomers now make up $\approx 16\%$ and the parallel component makes up 84% of the runtime at isomerspace C_{200}. Furthermore, the parallel components become ever larger constituents as N grows seeing as all the $\mathcal{O}(N^2)$ components run in lockstep parallel.

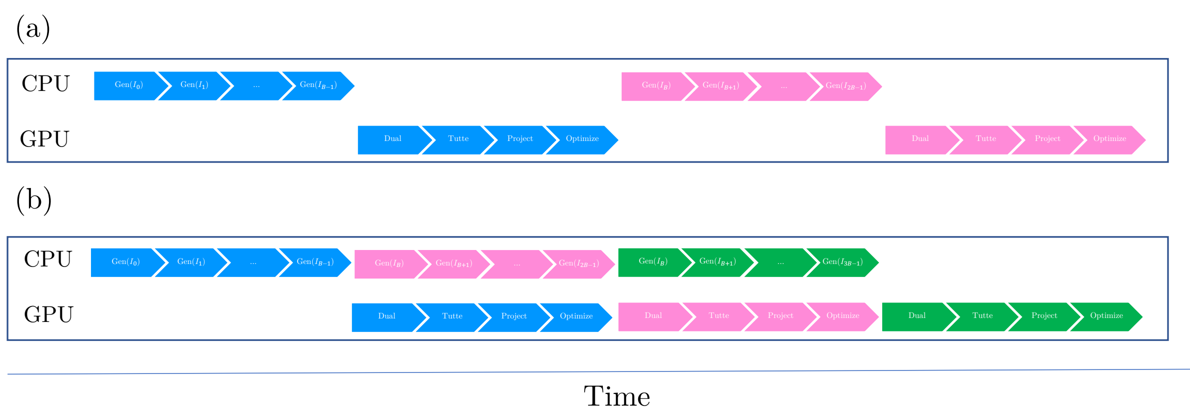


Figure 5.15: *Visualisation of (**a**) the sequential two-processing-stage pipeline and (**b**) a parallel pipeline implementation. Here the different colours signify different batches, **blue** **pink** and **green** are the first, second and third batches respectively.*

5.4 Pipeline 6: Pipeline Parallelism

Pipeline parallelism is the process of breaking a task into a sequence of processing stages, each of which depends on the one before it. However instead of waiting for each processing stage to finish before starting the next stage we can simply let each of the processing stages work on *different* problems simultaneously. In the context of pipeline 5, we first generate a full batch of isomers then insert these in an `IsomerQueue` after which the GPU performs the dualisation, Tutte embedding, spherical projection and forcefield optimisation, we then repeat this process until the entire isomerspace is completed. We realize that we can let the CPU produce isomers from `BuckyGen` while the GPU processes isomers present in the `IsomerQueue`. Fig. 5.15 shows how the sequential processing stages can be converted into a parallel pipeline.

The pipeline now works as follows: first produce an initial batch of isomers, insert these into an IsomerQueue, use this queue to refill The pipeline now consists of two separate processing stages that occur simultaneously. First we need to generate the initial batch, the blue batch in Fig. 5.15, then insert them into an `IsomerQueue`, call it **Input Queue**, which is used as a simple buffer. After the first set of isomers has been generated we refill an `IsomerBatch` with isomers. Now we can start the main loop where we first spawn a thread using std :: async() which then goes off and generates isomers while the GPU processing is initialised on the **Main Batch**, **Dualisation** $\rightarrow$ **Tutte Embedding** $\rightarrow$ **Spherical Projection** $\rightarrow$ **Forcefield Optimisation**. Once these steps complete we wait for the generating thread to complete, in the case that parallel processing surpasses generation, then refill the **Main Batch** with new isomers and repeat the loop until all isomers have been optimised. For our benchmark in Listing C.6 the same 680579 isomers sampled from isomerspace $C_{20}, C_{24}, \cdots, C_{200}$ are used, and the benchmark is executed 10 times for each isomerspace to account for fluctuations in runtime. The results are shown in Fig. 5.16.

We see that the pipeline parallelisation has almost perfectly amortised the runtime cost of generating isomers and converting their data layout, as the pipeline5 to pipeline4

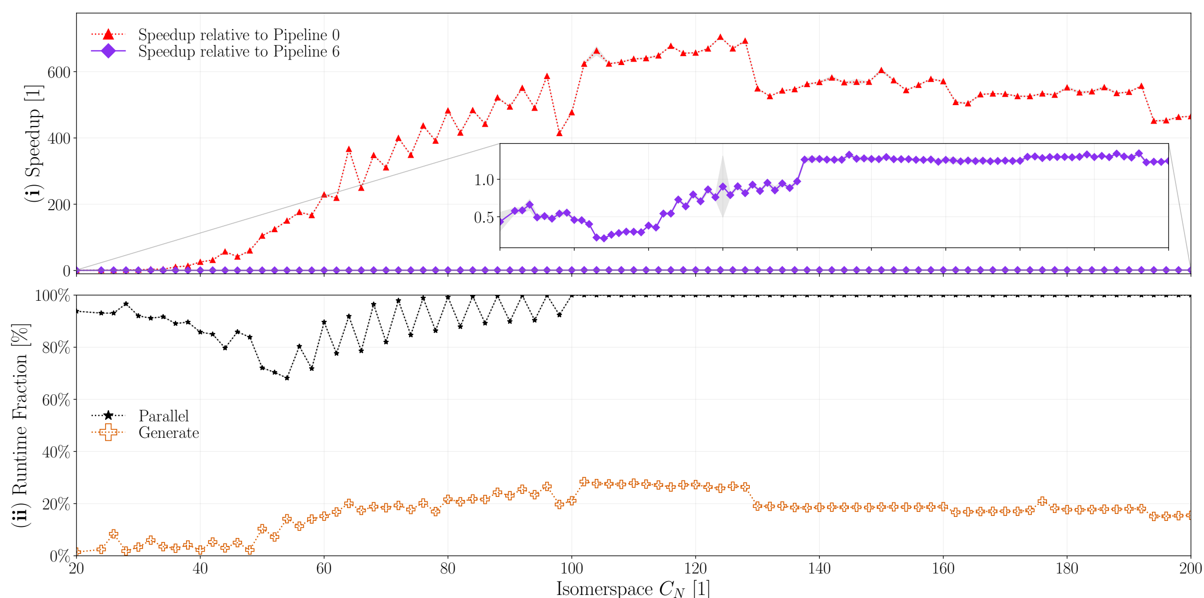


Figure 5.16: *(i) Speedup of pipeline 6 relative to the previous pipeline iteration, pipeline 5 and relative to the baseline pipeline 0. **ii** Fractional breakdown of the remaining runtime components, parallel ($\star$), generate and overhead 6. Shaded areas represent $\pm 2\sigma$.*

speedup closely matches the fractional cost that these components contributed in pipeline 5 (Fig. 5.14), e.g. at isomerspace C_{200} we expect a speedup of at most $1.16\times$ and indeed we achieve a speedup of $1.16(2)$ at this isomerspace, which is as much as we could hope for.

5.5 Pipeline 7: Multi-GPU Support

Amdahl's law states that the maximum theoretical speedup of an application is $S = 1/(1-P)$ where P is the parallel fraction of the program's runtime. Given the 84% parallel fraction of pipeline 6 we can predict that a maximum speedup of $S = 1/(1-0.84) = 6.25\times$ is possible. While traditionally this occurs as the number of parallel processing elements $p \to \infty$, however since we are able to hide the sequential component through pipeline parallelism and assuming that all parallel components scale perfectly it will ideally only require $84\%/16\% = 5.25\times$ more parallel hardware to achieve maximum throughput at isomerspace C_{200}. We now investigate how close to perfect scaling we are actually able to achieve by adding multi-GPU support.

With the tools developed so far, adding multi-GPU support is actually not very difficult using multiple `LaunchCtx` objects, we simply need to have all GPU functions switch to the device attributed to either a given `LaunchCtx` using cudaSetDevice(ctx.deivce). Furthermore, `IsomerBatch` and `IsomerQueue` now have a device attributed to them such that copying can be automatically handled using this information and cudaMemcpyPeerAsync() if we every wish to copy between GPUs. Now we just create two `IsomerBatches` (processing batches for each of the two GPUs), two `IsomerQueue` objects for insertion as

in pipeline 7. Importantly we use the `LaunchPolicy::ASYNC` to enqueue operations on both GPUs simultaneously from the main thread on the CPU without waiting for each to finish before enqueueing the other this is shown in listing. We note that we ought to however avoid this if possible since it just uses PCIE bandwidth as no GPU-GPU interconnect is present on the system we are benchmarking on.

Listing 5.1: *Excerpt from pipeline 7 benchmark script*

```
115     //Generate new isomers while processing the previous batch
116     auto generate_handle = std::async(std::launch::async, generate_isomers);
117     auto T2 = high_resolution_clock::now();
118       //Main processing
119       isomerspace_dual::dualise(B0, device0, LaunchPolicy::ASYNC);
120       isomerspace_dual::dualise(B1, device1, LaunchPolicy::ASYNC);
121       isomerspace_tutte::tutte_layout(B0, 10000000, device0, LaunchPolicy::ASYNC);
122       isomerspace_tutte::tutte_layout(B1, 10000000, device1, LaunchPolicy::ASYNC);
123       isomerspace_X0::zero_order_geometry(B0, 4.0, device0, LaunchPolicy::ASYNC);
124       isomerspace_X0::zero_order_geometry(B1, 4.0, device1, LaunchPolicy::ASYNC);
125       isomerspace_forcefield::optimise<PEDERSEN>(B0,N*5,N*5, device0, LaunchPolicy::ASYNC);
126       isomerspace_forcefield::optimise<PEDERSEN>(B1,N*5,N*5, device1, LaunchPolicy::ASYNC);
127       //Output finished isomers
128       Out_Q0.push(B0,device0,LaunchPolicy::ASYNC);
129       Out_Q1.push(B1,device1,LaunchPolicy::ASYNC);
130       device0.wait(); device1.wait();
131     auto T3 = high_resolution_clock::now(); T_par[l] += ( T3 - T2);
132       //Wait for generation to finish, if this process is faster than GPU operations, the
    call returns immediately.
133       generate_handle.wait();
134       //Refill batches with new isomers
135       In_Q0.refill_batch(B0, device0, LaunchPolicy::ASYNC);
136       In_Q1.refill_batch(B1, device1, LaunchPolicy::ASYNC);
137       device0.wait(); device1.wait();
138     auto T4 = high_resolution_clock::now(); T_io[l] += (T4 - T3);
139
140   }
```

Listing 5.1 demonstrates the use of different `LaunchCtx` objects to achieve concurrent multi-GPU utilisation, we benchmark this script using the same randomly sampled isomers as previously, but we synthetically insert each isomer in two different `IsomerQueue`, one for each GPU and process these as shown in the listing. Benchmarking isomerspaces $C_{20}, C_{24}, \cdots, C_{200}$ yields the results shown in Fig. 5.16. Seeing as the parallel component of the program previously constituted very close to 100% and the generation of isomers is still faster than the parallel components it is perhaps no surprise that we are able to reach close to perfect speedup i.e. doubling the amount of parallel processing power using 2 `RTX 3080` GPUs has yielded double the performance as evidenced by the benchmark. We reach a performance $900\times$ higher than that of the sequential pipeline and minimal overhead as both, data layout conversion and isomer generation can be hidden by pipeline parallelism.

5.6 Summary

In Section 5.1 we investigate the new bottleneck in the program *Tutte embedding*, which became visible after the massive speedup gained in the forcefield optimisation. We showed how the implementation of Tutte embedding in the *Fullerene* program was al-

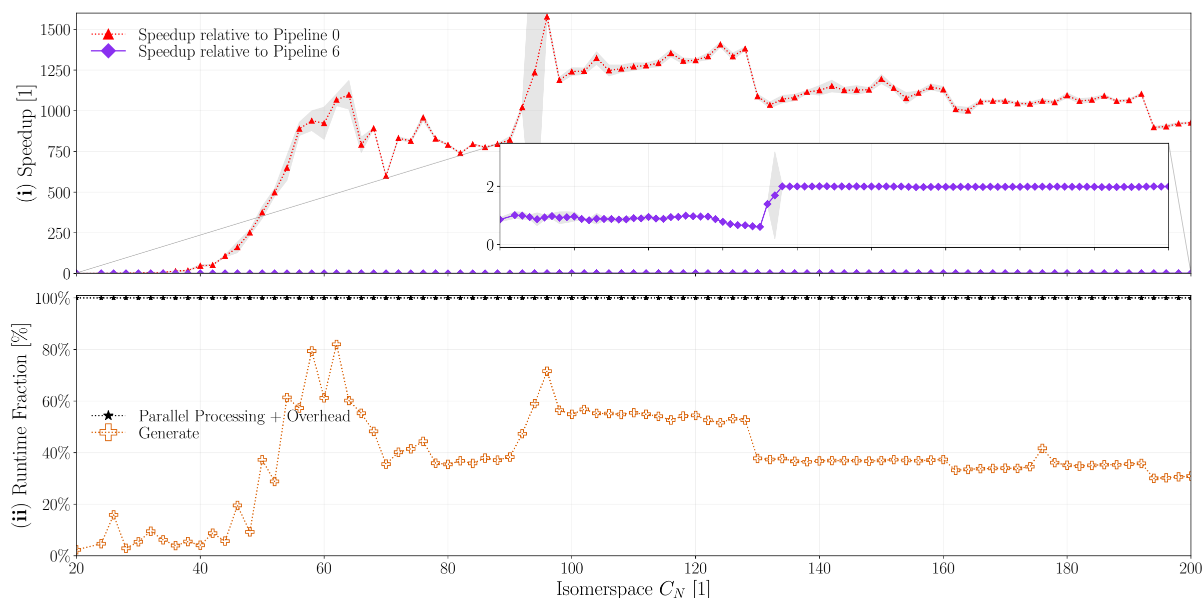


Figure 5.17: *(i) Speedup of pipeline 7 relative to the previous pipeline iteration, pipeline 7 and relative to the baseline pipeline 0.* ***ii*** *Fractional breakdown of the remaining runtime components, parallel ($\star$), generate and overhead 6. Shaded areas represent $\pm 2\sigma$.*

ready quite well suited for the lockstep parallel paradigm, since many of its components were readily data-parallel. We provided a fully lockstep-parallel method for Tutte embedding in Algorithm 21. Gauging the runtime complexity we noted slightly worse than quadratic scaling Fig. 5.5 presumably due to the use of a primitive solver. The CUDA/C++ implementation was nonetheless quite a bit faster than the sequential variant (1000x speedup). We evaluated the validity of our Tutte embedding by comparing the results directly to the sequential C++ implementation, we found that the vast majority of isomers had mean relative error of less than 10^{-5} (Fig. 5.3).

Having optimised the Tutte embedding the next bottleneck surfaced, spherical projection and the I/O associated with layout conversion (Fig. 5.6). We realised that the spherical projection algorithm present in the *Fullerene* program was not well suited for the lockstep parallel paradigm, since it employed a BFS traversal of the graph which requires dynamic parallelism at best. We therefore implemented lockstep parallelism, where immediately possible, and left the BFS traversal to be executed sequentially by a single GPU thread per isomer. The resulting spherical projection algorithm is shown in Algorithm 24 and was benchmarked to be 100-200x faster than the sequential version (Fig. 5.9). The validity of the spherical projection was verified by comparing the results to the sequential C++ implementation the same way the Tutte embedding was verified (Fig. 5.8).

Yet again, the removal of one bottleneck revealed a new one, this time the final sequential component (apart from the `BuckyGen` generator), the dualisation algorithm. We first go through the sequential implementation of the dualisation algorithm in Algorithm 25 we discover multiple problematic components, but first and foremost the use hash-maps

to store and access triangles and arcs. In Algorithm 26 we showed how we can circumvent the use of hash-maps entirely, instead using static arrays index juggling and an exclusive scan to achieve a massive 4000-6000x speedup (Fig. 5.12). The validation of the dualisation algorithm is much more direct, since the result is an adjacency-list consisting entirely of integers the results have to match the sequential implementation exactly, and they do.

At this point we had arrived at a total pipeline speedup of 350-500x, we saw that `BuckyGen` had started taking up a large fraction of the total runtime. This motivated us to pursue pipeline parallelism to overlap the generation of one batch of isomers, with the optimisation of another. Fig. 5.16 shows the results of this endeavour, we were able to achieve a total pipeline speedup of 450-700x. Finally, we explore the possibility of using multiple GPUs to further increase the speedup, we find that we can achieve a total speedup of 950-1400x (Fig. 5.17). A remarkable achievement considering the sequential implementation was written in Fortran and C++, both languages with strong performance characteristics.

Chapter 6

Conclusion and Future Work

6.1 Future Work

Production of isomer geometries is in itself interesting, but it serves a greater purpose, one of the main objectives in the CARMA project is to approximate the molecular properties of these isomers. Many of these properties require the Hessian to be calculated. Granted the speed of the pipeline presented in this thesis, the computation of the Hessian would need to be implemented in a comparatively efficient manner. As we alluded to in Section 4.2.2, the Hessian is a sparse matrix for our system, e.g. the bond-gradient $\nabla_a(R_{ab})$ depends only on the atoms a and b, therefore the hessian w.r.t to this term only has non-zero elements at the matrix-elements (a, a), (a, b). Preliminary work revealed that a total of 27 distinct analytical tensor expressions make up the Hessian for the current forcefield expression by Pedersen.[12] We have attempted to derive all of these expressions, but errors were made in the process due to the non-commutative nature of several tensor-tensor operations and ambiguous/mis- representation of tensor calculus on Wikipedia. Therefore, most of the derivations have to be redone in the future.

The flatness term warrants further investigation and comparison, of the acquired geometries to DFT calculations. This is necessary to evaluate the quality of the geometries and to extract accurate force-constants.

The Tutte-embedding (Algorithm 21) and spherical projection (Algorithm 24) algorithms presented in this work provide the efficient basis for generating initial geometries for the forcefield optimisation. Convergence of the forcefield relies heavily on the quality of these geometries therefore the Tutte embedding plus spherical projection is a candidate for substitution in the future. Indeed, more robust prototypes for producing accurate initial geometries have already been proposed by Avery. Work is required to adapt these to a node-parallelisable regime such that they can be implemented efficiently in lockstep parallel and compete computationally with the implementation in this thesis.

6.2 Conclusion

In this thesis we have developed and shown algorithms for lockstep parallelisation of forcefield optimisation for entire isomerspaces. In Section 3.4 we have shown algorithms and implementations of parallel primitives, scan and reduction, we saw how proper memory access pattern (Fig. 3.12) is critical to performance, a simple interchange of a slightly faster reduction method in the forcefield yielded 5% performance improvement (Fig. 4.18).

In Section 4.2 we showcased how a forcefield optimisation could be carried out in lockstep parallel, outperforming the previous state-of-the-art fullerene forcefield optimisation (Wirz et al.[18]) by a factor of 500-750x (Fig. 4.19).

In Section 4.5.6 we were able to develop a lockstep parallel approach to running variable number of iterations per isomer, by developing an efficient parallel queue (Fig. 4.13)

allowing us to extract an additional 15% performance (Fig. 4.15).

In Section 4.7.5 we demonstrated how we were able to add a flatness term to the harmonic energy expression and implement it in the lockstep-parallel forcefield without serious performance degradation. The flatness term, on internal-coordinate analysis, appears to trade bond-length deviation for face flatness. Visual inspection (Fig. 4.23) of select isomers found that the flatness term appears to improve convexity of isomers, this is desirable as fullerenes are known to be convex.

In Section 5.1 we showed how we were able to implement a Tutte embedding algorithm in lockstep parallel, achieving a speedup over the sequential C++ implementation from the *Fullerene* program, of 1000x (Fig. 5.4). Moreover, Section 5.2 saw a not fully parallel implementation of the spherical projection algorithm, achieving a speedup over the sequential C++ implementation from the *Fullerene* program, of 100x (Fig. 5.9).

In Section 5.3 we showed an extremely efficient dualisation algorithm which was the last piece of the pipeline-puzzle required to get proper speedup of the entire pipeline. The dualisation algorithm was implemented in a lockstep parallel way that requires zero hashmaps, thanks to clever indexing and efficient parallel exclusive scan primitives. Thus achieving a staggering speedup over the sequential C++ implementation from the *Fullerene* program, of 4000-6000x (Fig. 5.12).

In Section 5.4 we showed that with all the components in the pipeline parallelised we were able to achieve a total program speedup over the sequential pipeline, of 350-500x (Fig. 5.14). At this point we noted the `BuckyGen` generator had started to constitute a significant portion of the total runtime, we were able to hide this runtime by overlapping the `BuckyGen` and the lockstep parallel forcefield optimisation (pipeline parallelism) using asynchronous execution. This allowed us to achieve a total speedup over the sequential pipeline, of 450-700x (Fig. 5.16). Finally, we were able to demonstrate essentially perfect scaling from single to dual GPU achieving a total speedup over the sequential pipeline, of 950-1400x (Fig. 5.17).

Now, the main research question of this thesis was: *Can we use a lockstep parallel algorithm to exploit massively parallel hardware enabling exhaustive exploration of full isomerspaces?*. The final pipeline allows us exhaustively produce and optimise the entire C_{200} isomerspace in a projected time of 6 hours on two GPUs, making what was previously completely impractical (247 days), possible within the span of an afternoon. Thus, the answer to the question is a resounding yes.

Bibliography

[1] The house of graphs, fullerenes. https://houseofgraphs.org/meta-directory/fullerenes.

[2] Issue efficiency. https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/issueefficiency.htm.

[3] James Emil Avery. Face flatness measure and derivatives for forcefields. https://www.nbi.dk/~avery/CARMA/FF/flatness.pdf, 2022.

[4] Guy E. Blelloch. Scans as primitve parallel operations. *IEEE Transactions on computers (1989)*, 38(11), 1998.

[5] Gunnar Brinkmann, Jan Goedgebeur, and Brendan D. McKay. The Generation of Fullerenes. *Journal of Chemical Information and Modeling*, 52(11):2910–2918, November 2012. Publisher: American Chemical Society.

[6] Ulrich Drepper. What every programmer should know about memory. https://people.freebsd.org/~lstewart/articles/cpumemory.pdf, 2007.

[7] Mahdieh Hasheminezhad, Herbert Fleischner, and Brendan D. McKay. A universal set of growth operations for fullerenes. *Chemical Physics Letters*, 464(1):118–121, 2008.

[8] Carl-Johannes Johnsen. Automated ai learning for x-ray based classification, 2021.

[9] Harold Kroto. The stability of the fullerenes cn, with n = 24, 28, 32, 36, 50, 60 and 70. *Nature*, 329, 10 1987.

[10] Nishimura Takuji Matsumoto Makoto. Mersenne twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Transactions on Modeling and Computer Simulations (1998)*, 1998.

[11] Gordon E. Moore. *Understanding Moore's Law: Four Decades of Innovation*, chapter 7, pages 67–84. Chemical Heritage Foundation, 2006.

[12] Buster N. Pedersen. Molecular shapes of fullerenes - a robust force field method for fullerenes and polyhedral molecules, 2007.

[13] Larry Nazareth. A relationship between the bfgs and conjugate gradient algorithms and its implications for new algorithms. *SIAM Journal on Numerical Analysis*, 16(5):794–800, 1979.

[14] Buster P. Nielsen. Forcefield script by pedersen et al. https://github.com/Buster220992/Thesis-Molecular-Shapes-of-Fullerenes/blob/master/Fullerene_GeometryFunctions.py.

[15] Nvidia. Cuda occupancy calculator. https://docs.nvidia.com/cuda/cuda-occupancy-calculator/index.html, 2022.

[16] David Sehnal, Sebastian Bittrich, Mandar Deshpande, Radka Svobodová, Karel Berka, Václav Bazgier, Sameer Velankar, Stephen K Burley, Jaroslav Koča, and Alexander S Rose. Mol* Viewer: modern web app for 3D visualization and analysis of large biomolecular structures. *Nucleic Acids Research*, 49(W1):W431–W437, 05 2021.

[17] W. T. Tutte. How to draw a graph. 1962.

[18] Lukas Wirz, Ralf Tonner-Zech, Andreas Hermann, Rebecca Sure, and Peter Schwerdtfeger. From small fullerenes to the graphene limit: A harmonic force-field method for fullerenes and a comparison to density functional calculations for goldberg-coxeter fullerenes up to c-980. *Journal of Computational Chemistry*, 37:10, 01 2016.

[19] Z.C. Wu, Daniel A. Jelski, and Thomas F. George. Vibrational motions of buckminsterfullerene. *Chemical Physics Letters*, 137(3):291–294, 1987.

Chapter A

System Specifications

cpuinfo.txt

```
processor	: 0
vendor_id	: AuthenticAMD
cpu family	: 23
model		: 1
model name	: AMD Ryzen Threadripper 1950X 16-Core Processor
stepping	: 1
microcode	: 0x8001137
cpu MHz		: 2200.000
cache size	: 512 KB
physical id	: 0
siblings	: 32
core id		: 0
cpu cores	: 16
apicid		: 0
initial apicid	: 0
fpu		: yes
fpu_exception	: yes
cpuid level	: 13
wp		: yes
flags		: fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ht sys
bugs		: sysret_ss_attrs null_seg spectre_v1 spectre_v2 spec_store_bypass retbleed
bogomips	: 6799.32
TLB size	: 2560 4K pages
clflush size	: 64
cache_alignment	: 64
address sizes	: 43 bits physical, 48 bits virtual
power management: ts ttp tm hwpstate eff_freq_ro [13] [14]
```

meminfo.txt

```
Memory Device
	Array Handle: 0x000F
	Error Information Handle: 0x0016
	Total Width: 64 bits
	Data Width: 64 bits
	Size: 16 GB
	Form Factor: DIMM
	Set: None
	Locator: DIMM 0
	Bank Locator: P0 CHANNEL A
	Type: DDR4
	Type Detail: Synchronous Unbuffered Unregistered
	Speed: 2933 MT/s
	Manufacturer: Unknown
	Serial Number: 00000000
	Asset Tag: Not Specified
	Part Number: F4-3200C14-16GTZR
	Rank: 2
	Configured Memory Speed: 1467 MT/s
	Minimum Voltage: 1.2 V
	Maximum Voltage: 1.2 V
	Configured Voltage: 1.2 V
```

free.txt

```
              total        used        free      shared  buff/cache   available
Mem:          125Gi        13Gi       7.7Gi       256Mi       104Gi       110Gi
Swap:         8.0Gi       877Mi       7.1Gi
```

device_info.txt

```
Device Number: 0
Device name: NVIDIA GeForce RTX 3080
Memory Clock Rate KHz: 9501000
Clock Rate KHz: 1740000
Memory Bus Width bits: 320
Peak Memory Bandwidth GB/s: 760.080000
L2 Cache Size bytes: 5242880
Total Global Memory bytes: 1914765312
Shared Memory Per Block bytes: 49152
Registers Per Block: 65536
Registers Per Multiprocessor: 65536
Warp Size: 32
Cooperative Launch Support: 1
Concurrent Kernels Support: 1
Max Threads Per Block: 1024
Max Threads Per Multiprocessor: 1536
Max Threads Per Dimension: 1024, 1024, 64
Max Grid Size: 2147483647, 65535, 65535
Reserved Shared Memory Per Block bytes: 49152
Single to Double Precision Performance Ratio: 32
Compute Capability: 8.6
Number of Multiprocessors: 68

Device Number: 1
Device name: NVIDIA GeForce RTX 3080
Memory Clock Rate KHz: 9501000
Clock Rate KHz: 1740000
Memory Bus Width bits: 320
Peak Memory Bandwidth GB/s: 760.080000
L2 Cache Size bytes: 5242880
Total Global Memory bytes: 1911488512
Shared Memory Per Block bytes: 49152
Registers Per Block: 65536
Registers Per Multiprocessor: 65536
Warp Size: 32
Cooperative Launch Support: 1
Concurrent Kernels Support: 1
Max Threads Per Block: 1024
Max Threads Per Multiprocessor: 1536
Max Threads Per Dimension: 1024, 1024, 64
Max Grid Size: 2147483647, 65535, 65535
Reserved Shared Memory Per Block bytes: 49152
Single to Double Precision Performance Ratio: 32
Compute Capability: 8.6
Number of Multiprocessors: 68
```

A.1 Compilers

For this thesis we used the following compilers to compile Fortran, C++ and CUDA code respectively:

- gfortran 9.5.0
- g++ 11.3.0
- nvcc 11.8.89

Chapter B

Implementation Code

Listing B.1: *Graph Utility*

```
__device__ device_node_t dedge_ix(const device_node_t u, const device_node_t v) const{
    for (uint8_t j = 0; j ¡ 3; j++)
        if (cubic_neighbours[u*3 + j] == v) return j;

    assert(false);
    return 0;               // Make compiler happy
}
__device__ device_node_t next(const device_node_t u, const device_node_t v) const{
    device_node_t j = dedge_ix(u,v);
    return cubic_neighbours[u*3 + ((j+1)%3)];
}
__device__ device_node_t prev(const device_node_t u, const device_node_t v) const{
    device_node_t j = dedge_ix(u,v);
    return cubic_neighbours[u*3 + ((j+2)%3)];
}
__device__ device_node_t next_on_face(const device_node_t u, const device_node_t v) const{
    return prev(v,u);
}
__device__ device_node_t prev_on_face(const device_node_t u, const device_node_t v) const{
    return next(v,u);
}
__device__ device_node_t face_size(device_node_t u, device_node_t v) const{
    device_node_t d = 1;
    device_node_t u0 = u;
    while (v != u0)
    {
        device_node_t w = v;
        v = next_on_face(u,v);
        u = w;
        d++;
    }
    return d;
}
__device__ uint8_t get_face_oriented(device_node_t u, device_node_t v, device_node_t *f) const{
    constexpr int f_max = 6;
    int i = 0;
        f[0] = u;
    while (v!=f[0] && i¡f_max)
    {
        i++;
        device_node_t w = next_on_face(u,v);
        f[i] = v;
        u = v;
        v = w;
    }
    if(i>=f_max) {assert(false); return 0;} //Compiler wants a return statement
    else return i + 1;
}
__device__ device_node2 get_face_representation(device_node_t u, device_node_t v) const{
    constexpr int f_max =6;
    int i = 0;
    auto start_node = u;
    device_node2 min_edge = {u,v};
    while (v!= start_node && i ¡ f_max){
        device_node_t w = next_on_face(u, v);
        u = v; v = w;
        if(u ¡ min_edge.x) min_edge = {u,v};
       ++i;
    }
    //assert(next_on_face(u,v) == start_node);
    return min_edge;
}
```

Listing B.2: *Constants*

```
struct Constants{
    device_coord3d f_bond;
    device_coord3d f_inner_angle;
    device_coord3d f_inner_dihedral;
    device_coord3d f_outer_angle_m;
    device_coord3d f_outer_angle_p;
    device_coord3d f_outer_dihedral;
    device_real_t f_flat = 1e2;

    device_coord3d r0;
    device_coord3d angle0;
    device_coord3d outer_angle_m0;
    device_coord3d outer_angle_p0;
    device_coord3d inner_dih0;
    device_coord3d outer_dih0_a;
    device_coord3d outer_dih0_m;
    device_coord3d outer_dih0_p;
    INLINE Constants(const IsomerBatch& G, const uint32_t isomer_idx){
        //Set pointers to start of fullerene.
        const DeviceCubicGraph FG(&G.cubic_neighbours[isomer_idx*blockDim.x*3]);
        device_node3 cubic_neighbours = {FG.cubic_neighbours[threadIdx.x*3],
                                         FG.cubic_neighbours[threadIdx.x*3 + 1],
                                         FG.cubic_neighbours[threadIdx.x*3 + 2]};
        for (uint8_t j = 0; j ¡ 3; j++) {
            //Faces to the right of arcs ab, ac and ad.

            uint8_t F1 = FG.face_size(threadIdx.x, d_get(cubic_neighbours, j)) - 5;
            uint8_t F2 = FG.face_size(threadIdx.x, d_get(cubic_neighbours, (j+1)%3)) -5;
            uint8_t F3 = FG.face_size(threadIdx.x, d_get(cubic_neighbours, (j+2)%3)) -5;

            //The faces to the right of the arcs ab, bm and bp in no particular order, from this we can ded
            uint8_t neighbour_F1 = FG.face_size(d_get(cubic_neighbours, j),
                                   FG.cubic_neighbours[d_get(cubic_neighbours, j)*3] ) -5;
            uint8_t neighbour_F2 = FG.face_size(d_get(cubic_neighbours, j),
                                   FG.cubic_neighbours[d_get(cubic_neighbours, j)*3 + 1] ) -5;
            uint8_t neighbour_F3 = FG.face_size(d_get(cubic_neighbours, j),
                                   FG.cubic_neighbours[d_get(cubic_neighbours, j)*3 + 2] ) -5;

            uint8_t F4 = neighbour_F1 + neighbour_F2 + neighbour_F3 - F1 - F3 ;
            d_set(r0,               j,  optimal_bond_lengths[F3 + F1]);
            d_set(angle0,           j,  optimal_corner_cos_angles[F1]);
            d_set(inner_dih0,       j,  optimal_dih_cos_angles[face_index(F1, F2 , F3)]);
            d_set(outer_angle_m0,   j,  optimal_corner_cos_angles[F3]);
            d_set(outer_angle_p0,   j,  optimal_corner_cos_angles[F1]);
            d_set(outer_dih0_a,     j,  optimal_dih_cos_angles[face_index(F3, F4, F1)]);
            d_set(outer_dih0_m,     j,  optimal_dih_cos_angles[face_index(F4, F1, F3)]);
            d_set(outer_dih0_p,     j,  optimal_dih_cos_angles[face_index(F1, F3, F4)]);

            //Load force constants from neighbouring face information.
            d_set(f_bond,           j,  bond_forces[F3 + F1]);
            d_set(f_inner_angle,    j,  angle_forces[F1]);
            d_set(f_inner_dihedral, j,  dih_forces[F1 + F2 + F3]);
            d_set(f_outer_angle_m,  j,  angle_forces[F3]);
            d_set(f_outer_angle_p,  j,  angle_forces[F1]);
            d_set(f_outer_dihedral, j,  dih_forces[F1 + F3 + F4]);
            #endif
        }
    }
};
```

Listing B.3: *Queue Resizing*

```
cudaError_t IsomerQueue::resize(const size_t new_capacity,const LaunchCtx& ctx, const LaunchPolicy policy){
    cudaSetDevice(m_device);
    //Lambda function because it is only called in here and avoids duplication of code because these transformati
    auto queue_resize_batch = [&](IsomerBatch& batch){
        //Construct a tempory batch: allocates the needed amount of memory.
        IsomerBatch temp_batch = IsomerBatch(batch.n_atoms, new_capacity, batch.buffer_type, batch.get_device_id(
        //Copy contents of old batch into newly allocated memory.
        cuda_io::copy(temp_batch,batch,ctx,policy,
                      {0, batch.isomer_capacity - *props.front}, {*props.front,batch.isomer_capacity});
        cuda_io::copy(temp_batch,batch,ctx,policy,
                      {batch.isomer_capacity- *props.front, batch.isomer_capacity - *props.front + *props.back},
                      {0,*props.back});
        for (int i = 0; i ¡ batch.pointers.size(); i++)
        {
            void* temp_ptr = *get¡1>(batch.pointers[i]);
            printLastCudaError("Free␣failed");
            //Reassign pointers of the input batch, to the new memory
            *get¡1>(batch.pointers[i]) = *get¡1>(temp_batch.pointers[i]);
            //Assign old pointers to temporary object, let destructor take care of cleanup.
            *get¡1>(temp_batch.pointers[i]) = temp_ptr;
        }
        batch.isomer_capacity = temp_batch.isomer_capacity;
    };
    if (*props.back >= *props.front){
        cuda_io::resize(host_batch, new_capacity, ctx, policy);
        cuda_io::resize(device_batch, new_capacity, ctx, policy);
    }else{
        queue_resize_batch(host_batch);
        queue_resize_batch(device_batch);
        *props.front = 0;
        *props.back  = *props.size - 1;
    }
    *props.capacity = new_capacity;
    is_host_updated = true;
    is_device_updated = true;
    return cudaGetLastError();
}
```

Listing B.4: *Queue Pop*

```
Polyhedron IsomerQueue::pop(const LaunchCtx& ctx, const LaunchPolicy policy){
    cudaSetDevice(m_device);
    to_host(ctx);
    if(*props.size == 0) throw std::runtime_error("Queue is empty");
    neighbours_t out_neighbours(N, std::vector<int>(3));
    for(int i = 0; i < N; i++){
        out_neighbours[i][0] = host_batch.cubic_neighbours[(*props.front)*N*3 + i*3 + 0];
        out_neighbours[i][1] = host_batch.cubic_neighbours[(*props.front)*N*3 + i*3 + 1];
        out_neighbours[i][2] = host_batch.cubic_neighbours[(*props.front)*N*3 + i*3 + 2];
    }
    std::vector<coord3d> out_coords(N);
    for(int i = 0; i < N; i++){
        out_coords[i][0] = host_batch.X[(*props.front)*N*3 + i*3 + 0];
        out_coords[i][1] = host_batch.X[(*props.front)*N*3 + i*3 + 1];
        out_coords[i][2] = host_batch.X[(*props.front)*N*3 + i*3 + 2];
    }
    *props.front = (*props.front + 1) % *props.capacity;
    *props.size -= 1;
    return Polyhedron(Graph(out_neighbours,true),out_coords);
}
```

Listing B.5: *Single Isomer Push*

```
cudaError_t IsomerQueue::insert(const Graph& in, const size_t ID, const LaunchCtx& ctx, const LaunchPolicy policy
    cudaSetDevice(m_device);
    //Before inserting a new isomer, make sure that the host batch is up to date with the device version.
    to_host(ctx);

    //If the queue is full, double the size, same beahvior as dynamically allocated containers in the std library
    if (*props.capacity == *props.size){
        resize(*props.capacity * 2, ctx, policy);
    }

    //Edge case: queue has no members
    if(*props.back == -1){
        *props.front = 0;
        *props.back = 0;
    } else{
    *props.back = *props.back + 1 % *props.capacity;}

    //Extract the graph information (neighbours) from the PlanarGraph object and insert it at the appropriate loc
    auto Nf = N / 2 + 2;
    size_t face_offset = *props.back * Nf;
    for(node_t u=0;u<in.neighbours.size();u++){
        host_batch.face_degrees[face_offset + u] = in.neighbours[u].size();
        for(int j=0;j<in.neighbours[u].size();j++){
            host_batch.dual_neighbours[6*(face_offset+u)+j] = in.neighbours[u][j];
        }
    }


    //Assign metadata.
    host_batch.statuses[*props.back] = IsomerStatus::NOT_CONVERGED;
    host_batch.IDs[*props.back] = ID;
    host_batch.iterations[*props.back] = 0;
    *props.size += 1;

    //Since the host batch has been updated the device is no longer up to date.
    is_device_updated = false;
    return cudaGetLastError();
}
```

Listing B.6: *Refill Batch*

```
__global__ void refill_batch_(IsomerBatch B, IsomerBatch Q_B, IsomerQueue::QueueProperties queue,
int* scan_array){
    auto num_inside_circular_range = [](const int begin, const int end, const int test_val){
        if(begin ¡ 0  end ¡ 0) {return false;}
        if (begin ¡= end) {return (test_val >= begin && test_val ¡= end);}
        else {return (test_val >= begin  test_val ¡= end);}
    };

    //Must ensure that all writes to queue counters from the host are visible to the device threads before
    __threadfence_system();
    DEVICE_TYPEDEFS
    extern __shared__ int smem[];
    auto Nf = B.n_faces; // Number of faces
    auto queue_requests = 0;
    //Grid stride for loop, allows for handling of any batch size.
    auto limit = ((B.isomer_capacity + gridDim.x - 1) / gridDim.x ) * gridDim.x;  //Fast ceiling integer di
    for (int isomer_idx = blockIdx.x; isomer_idx ¡ limit; isomer_idx+= gridDim.x){
    GRID_SYNC
    bool access_queue = false;
    if (isomer_idx ¡ B.isomer_capacity) access_queue = B.statuses[isomer_idx] != IsomerStatus::NOT_CONVERGE
    //Perform a grid scan over the array of 0s and 1s (1 if a block needs to access the queue and 0 otherwi
    //If the grid is larger than the batch capacity, we only want to scan over the interval [0,B.isomer_cap
    grid_ex_scan¡int>(scan_array, smem, (int)access_queue, min((int)B.isomer_capacity, (int)gridDim.x) + 1)
    //The block gets its uniqueue index from the back of the queue plust its position in the scan array.
    auto queue_index = (*queue.front + scan_array[blockIdx.x] + queue_requests) % *queue.capacity;
    //Check that the index is inside the queue
    access_queue &= num_inside_circular_range(*queue.front, *queue.back, queue_index);
    if (access_queue){
        //Given the queue index, copy data from the queue (container Q_B) to the target batch B.
        size_t queue_array_idx     = queue_index*blockDim.x+threadIdx.x;
        size_t global_idx          = blockDim.x*isomer_idx + threadIdx.x;
        reinterpret_cast¡coord3d*>(B.X)[global_idx]                 = reinterpret_cast¡coord3d*>(Q_B.X)[que
        reinterpret_cast¡node3*>(B.cubic_neighbours)[global_idx]     = reinterpret_cast¡node3*>(Q_B.cubic_ne
        reinterpret_cast¡coord2d*>(B.xys)[global_idx]                = reinterpret_cast¡coord2d*>(Q_B.xys)[q
        //Face parallel copying
        if (threadIdx.x ¡ Nf){
            size_t queue_face_idx     = queue_index* Nf + threadIdx.x;
            size_t output_face_idx    = isomer_idx * Nf + threadIdx.x;
            reinterpret_cast¡node6*>(B.dual_neighbours)[output_face_idx]            = reinterpret_cast¡n
            reinterpret_cast¡uint8_t*>(B.face_degrees)[output_face_idx]             = reinterpret_cast¡u
        }
        //Per isomer meta data
        if (threadIdx.x == 0){
            B.IDs[isomer_idx] = Q_B.IDs[queue_index];
            B.iterations[isomer_idx] = 0;
            B.statuses[isomer_idx] = Q_B.statuses[queue_index];
            Q_B.statuses[queue_index] = IsomerStatus::EMPTY;
        }
    }
    queue_requests += scan_array[min((int)B.isomer_capacity, (int)gridDim.x)]; //The last element of the sc
    }
    GRID_SYNC
    if ((threadIdx.x + blockIdx.x) == 0) {
        //Main thread checks if the number of requests for new isomers is greater than the queue size, then
        bool enough_left_in_queue = *queue.size >= queue_requests;
        atomicSub_system(queue.size, enough_left_in_queue ? queue_requests : *queue.size);
        if (*queue.size == 0) {
            atomicExch_system(queue.front,-1);
            atomicExch_system(queue.back, -1);
            } else{
            //Here we correct the front index by considering, what the front was at the start, how many req
            atomicExch_system(queue.front,
            enough_left_in_queue ? (*queue.front + queue_requests) % *queue.capacity  : *queue.back);
        }
    }
}
```

Listing B.7: *Batch Push*

```
__global__ void push_(IsomerBatch B, IsomerBatch Q_B, IsomerQueue::QueueProperties queue,  int* scan_array){
    auto num_inside_capacity = [](const int size, const int capacity, const int test_val){
        if(size ¡ 0  capacity ¡ 0  test_val ¡ 0) {return false;}
        return size + test_val ¡= capacity;
    };
    //Must ensure that all writes to queue counters from the host are visible to the device threads before readin
    __threadfence_system();
    DEVICE_TYPEDEFS
    extern __shared__ int smem[];
    auto Nf = B.n_faces; // Number of faces
    auto queue_requests = 0;
    //Grid stride for loop, allows for handling of any batch size.
    auto limit = ((B.isomer_capacity + gridDim.x - 1) / gridDim.x ) * gridDim.x;  //Fast ceiling integer division
    for (int isomer_idx = blockIdx.x; isomer_idx ¡ limit; isomer_idx+= gridDim.x){
    GRID_SYNC
    bool access_queue = false;
    if (isomer_idx ¡ B.isomer_capacity) access_queue = (B.statuses[isomer_idx] == IsomerStatus::CONVERGED)  (B.st
IsomerStatus::FAILED);
    grid_ex_scan¡int>(scan_array, smem, (int)access_queue, min((int)B.isomer_capacity, (int)gridDim.x) + 1);
    //Checks if the isomer owned by a given block is finished or empty, if it is we want to replace it with a new
    int queue_index = *queue.back ¡ 0 ? (scan_array[blockIdx.x] + queue_requests) : (*queue.back + 1 + queue_requ
    access_queue &= num_inside_capacity(*queue.size, *queue.capacity, queue_requests + scan_array[blockIdx.x]);
    if (access_queue){
        //Given the queue index, copy data from the queue (container Q_B) to the target batch B.
        size_t queue_array_idx    = queue_index*blockDim.x+threadIdx.x;
        size_t global_idx         = blockDim.x*isomer_idx + threadIdx.x;
        reinterpret_cast¡coord3d*>(Q_B.X)[queue_array_idx]                = reinterpret_cast¡coord3d*>(B.X)[global_
        reinterpret_cast¡node3*>(Q_B.cubic_neighbours)[queue_array_idx] = reinterpret_cast¡node3*>(B.cubic_neighb
        reinterpret_cast¡coord2d*>(Q_B.xys)[queue_array_idx]            = reinterpret_cast¡coord2d*>(B.xys)[globa
        //Face parallel copying
        if (threadIdx.x ¡ Nf){
            size_t queue_face_idx     = queue_index* Nf + threadIdx.x;
            size_t output_face_idx    = isomer_idx * Nf + threadIdx.x;
            reinterpret_cast¡node6*>(Q_B.dual_neighbours)[queue_face_idx] = reinterpret_cast¡node6*>(B.dual_neigh
            reinterpret_cast¡uint8_t*>(Q_B.face_degrees)[queue_face_idx]  = reinterpret_cast¡uint8_t*>(B.face_deg
        }
        //Per isomer meta data
        if (threadIdx.x == 0){
            Q_B.IDs[queue_index] = B.IDs[isomer_idx];
            Q_B.iterations[queue_index] = B.iterations[isomer_idx];
            Q_B.statuses[queue_index] = B.statuses[isomer_idx];
            B.statuses[isomer_idx] = IsomerStatus::EMPTY;
        }
    }
    queue_requests += scan_array[min((int)B.isomer_capacity, (int)gridDim.x)]; //The last element of the scanned
    }
    GRID_SYNC
    if ((threadIdx.x + blockIdx.x) == 0) {
        if (*queue.size == 0 && queue_requests > 0) {
            atomicExch_system(queue.front, 0); atomicExch_system(queue.back, queue_requests-1);}
        else{
            atomicExch_system(queue.back, (*queue.back + queue_requests) % *queue.capacity);
        }
        //Pushing to queue simply increases the size by the number of push requests.
        atomicAdd_system(queue.size, queue_requests);
    }
}
```

Listing B.8: *ArcData Implementation*

```
struct ArcData{
    //124 FLOPs;
    uint8_t j;
    INLINE ArcData(const uint8_t j, const coord3d* __restrict__ X, const NodeNeighbours& G){
        __builtin_assume(j < 3);
        this->j = j;
        node_t a = threadIdx.x;
        real_t r_rmp;
        coord3d ap, am, ab, ac, ad, mp;
        coord3d X_a = X[a], X_b = X[d_get(G.cubic_neighbours,j)];

        //Compute the arcs ab, ac, ad, bp, bm, ap, am, mp, bc and cd
        ab = (X_b - X_a);  r_rab = bond_length(ab); ab_hat = r_rab * ab;
        ac = (X[d_get(G.cubic_neighbours,(j+1)%3)] - X_a); r_rac = bond_length(ac); ac_hat = r_rac *
ac; rab = non_resciprocal_bond_length(ab);
        ad = (X[d_get(G.cubic_neighbours,(j+2)%3)] - X_a); r_rad = bond_length(ad); ad_hat = r_rad *
ad;

        coord3d bp = (X[d_get(G.next_on_face,j)] - X_b); bp_hat = unit_vector(bp);
        coord3d bm = (X[d_get(G.prev_on_face,j)] - X_b); bm_hat = unit_vector(bm);

        ap = bp + ab; r_rap = bond_length(ap); ap_hat = r_rap * ap;
        am = bm + ab; r_ram = bond_length(am); am_hat = r_ram * am;
        mp = bp - bm; r_rmp = bond_length(mp); mp_hat = r_rmp * mp;

        bc_hat = unit_vector(ac - ab);
        cd_hat = unit_vector(ad - ac);

        //Compute inverses of some arcs, these are subject to be omitted if the equations are adapted appr
        ba_hat = -ab_hat;
        mb_hat = -bm_hat;
        pa_hat = -ap_hat;
        pb_hat = -bp_hat;
    }
    INLINE real_t harmonic_energy(const real_t p0, const real_t p) const{
        return (real_t)0.5*(p-p0)*(p-p0);
    }
    INLINE coord3d  harmonic_energy_gradient(const real_t p0, const real_t p, const coord3d gradp) const{
        return (p-p0)*gradp;
    }
    INLINE real_t bond() const {return rab;}
    INLINE real_t angle() const {return dot(ab_hat,ac_hat);}
    INLINE real_t dihedral() const
    {
        coord3d nabc, nbcd; real_t cos_b, cos_c, r_sin_b, r_sin_c;
        cos_b = dot(ba_hat,bc_hat);  r_sin_b = (real_t)1.0/sqrt((real_t)1.0 - cos_b*cos_b); nabc =
cross(ba_hat, bc_hat) * r_sin_b;
        cos_c = dot(-bc_hat,cd_hat); r_sin_c = (real_t)1.0/sqrt((real_t)1.0 - cos_c*cos_c); nbcd =
cross(-bc_hat,cd_hat) * r_sin_c;
        return dot(nabc, nbcd);
    }
    coord3d
```

Chapter C

Benchmark Scripts

Listing C.1: *Benchmark Pipeline V0 Script*

```
#include "fullerenes/gpu/isomer_queue.hh"
#include "fullerenes/gpu/cuda_io.hh"
#include "fullerenes/gpu/kernels.hh"

const std::unordered_map¡size_t,size_t> num_fullerenes =
    ↪ {{20,1},{22,0},{24,1},{26,1},{28,2},{30,3},{32,6},{34,6},{36,15},{38,17},{40,40},{42,45},{44,89},{46,116},{48,1
    ↪ ;
using namespace chrono;
using namespace chrono_literals;

int main(int argc, char** argv){
    const size_t N_start                = argc>1 ? strtol(argv[1],0,0) : 20;
        ↪                    // Argument 1: Number of vertices N
    const size_t N_limit                = argc>2 ? strtol(argv[2],0,0) : 200;
        ↪                   // Argument 1: Number of vertices N

    auto N_runs = 1;
    ofstream out_file("IsomerspaceOpt_V0_" + to_string(N_limit) + ".txt");
    ofstream out_file_std("IsomerspaceOpt_V0_STD_" + to_string(N_limit) + ".txt");
    for (size_t N = N_start; N ¡ N_limit+1; N+=2)
    {
        if (N == 22) continue;
        auto sample_size = min(200,(int)num_fullerenes.find(N)->second);
        BuckyGen::buckygen_queue  Q  = BuckyGen::start(N,false,false);
        auto Nf = N/2 + 2;
        Graph FF;
        FF.neighbours = neighbours_t(Nf, std::vector¡node_t>(6));
        FullereneDual G;
        G.neighbours = neighbours_t(Nf, std::vector¡node_t>(6));
        G.N = Nf;
        bool more_to_generate = true;

        auto T0 = high_resolution_clock::now();
        auto

            T_seq     = std::vector¡std::chrono::nanoseconds>(N_runs),
            T_par     = std::vector¡std::chrono::nanoseconds>(N_runs),
            T_io      = std::vector¡std::chrono::nanoseconds>(N_runs);

        auto path = "isomerspace_samples/dual_layout_" + to_string(N) + "_seed_42";
        ifstream isomer_sample(path,std::ios::binary);
        auto fsize = std::filesystem::file_size(path);
        std::vector¡device_node_t> input_buffer(fsize/sizeof(device_node_t));
        auto available_samples = fsize / (Nf*6*sizeof(device_node_t));
        isomer_sample.read(reinterpret_cast¡char*>(input_buffer.data()), Nf*6*sizeof(
            ↪ device_node_t)*available_samples);

        std::vector¡int> random_IDs(available_samples);
        std::iota(random_IDs.begin(), random_IDs.end(), 0);
        std::shuffle(random_IDs.begin(), random_IDs.end(), std::mt19937{42});
        std::vector¡int> id_subset(random_IDs.begin(), random_IDs.begin()+sample_size);
        bool more = true;
        for (int l = 0; l ¡ N_runs; l++){
        for (int i = 0; i ¡ sample_size; ++i){
                for (size_t j = 0; j ¡ Nf; j++){
                    if(more) more &= BuckyGen::next_fullerene(Q, FF);
                    G.neighbours[j].clear();
                    for (size_t k = 0; k ¡ 6; k++) {
                        auto u = input_buffer[id_subset[i]*Nf*6 + j*6 +k];
                        if(u != UINT16_MAX) G.neighbours[j].push_back(u);
                    }
                }
                    auto T0 = high_resolution_clock::now();
                G.update();
                PlanarGraph pG = G.dual_graph();
                pG.layout2d  = pG.tutte_layout();
```

```
                auto T1 = high_resolution_clock::now(); T_seq[l] += T1 - T0;
            Polyhedron P(pG);
                auto T2 = high_resolution_clock::now(); T_io[l] += T2 - T1;
            P.points    = P.zero_order_geometry();
            P.optimize();
                auto T3 = high_resolution_clock::now(); T_seq[l] += T3 - T2;

    }}
    using namespace cuda_io;
    auto total = (float)(mean(T_io)/1ns + mean(T_par)/1ns + mean(T_seq)/1ns);
    std::cout << std::fixed << std::setprecision(2) << N << ",␣"<< sample_size << ",␣" <<
        ↪ (mean(T_seq)/1ns)/total*100. << "%,␣" << (mean(T_par)/1ns)/total*100. << "%,␣"
        ↪ << (mean(T_io)/1ns)/total*100. << "%,␣" << (float)(mean(T_io)/1us+mean(T_par)/1
        ↪ us+mean(T_seq)/1us)/sample_size << "us/isomer\n";
    out_file << N << ",␣"<< sample_size << ",␣" << mean(T_seq)/1ns << ",␣" << mean(T_par)
        ↪ /1ns <<  ",␣" << mean(T_io)/1ns << "\n";
    out_file_std << N << ",␣"<< sample_size << ",␣" << sdev(T_seq)/1ns << ",␣" << sdev(
        ↪ T_par)/1ns <<  ",␣" << sdev(T_io)/1ns << "\n";
  }

}
```

Listing C.2: *Benchmark Pipeline V1 Script*

```
#include ”fullerenes/gpu/isomer_queue.hh”
#include ”fullerenes/gpu/cuda_io.hh”
#include ”fullerenes/gpu/kernels.hh”

const std::unordered_map¡size_t,size_t> num_fullerenes =
    ↪ {{20,1},{22,0},{24,1},{26,1},{28,2},{30,3},{32,6},{34,6},{36,15},{38,17},{40,40},{42,45},{44,89},{46,116},{48,1
    ↪ ;
using namespace chrono;
using namespace chrono_literals;

int main(int argc, char** argv){
    const size_t N_start                = argc>1 ? strtol(argv[1],0,0) : 20;
        ↪                    // Argument 1: Number of vertices N
    const size_t N_limit                = argc>2 ? strtol(argv[2],0,0) : 200;
        ↪                   // Argument 1: Number of vertices N

    auto N_runs = 10;

    ofstream out_file(”IsomerspaceOpt_V1_” + to_string(N_limit) + ”.txt”);
    ofstream out_file_std(”IsomerspaceOpt_V1_STD_” + to_string(N_limit) + ”.txt”);
    for (size_t N = N_start; N ¡ N_limit+1; N+=2)
    {
        if (N == 22) continue;
        auto sample_size = min(gpu_kernels::isomerspace_forcefield::optimal_batch_size(N,0)*
            ↪ 4,(int)num_fullerenes.find(N)->second);
        auto M_b = min(gpu_kernels::isomerspace_forcefield::optimal_batch_size(N,0),(int)
            ↪ num_fullerenes.find(N)->second);

        //Pre allocate the device queue such that it doesn't happen during benchmarking
        auto Nf = N/2 + 2;
        FullereneDual G;
        G.neighbours = neighbours_t(Nf, std::vector¡node_t>(6));
        G.N = Nf;
        bool more_to_generate = true;

        auto T0 = high_resolution_clock::now();
        auto

            T_seq     = std::vector¡std::chrono::nanoseconds>(N_runs),
            T_par     = std::vector¡std::chrono::nanoseconds>(N_runs),
            T_io      = std::vector¡std::chrono::nanoseconds>(N_runs);

        auto path = ”isomerspace_samples/dual_layout_” + to_string(N) + ”_seed_42”;
        ifstream isomer_sample(path,std::ios::binary);
        auto fsize = std::filesystem::file_size(path);
        std::vector¡device_node_t> input_buffer(fsize/sizeof(device_node_t));
        auto available_samples = fsize / (Nf*6*sizeof(device_node_t));
        isomer_sample.read(reinterpret_cast¡char*>(input_buffer.data()), Nf*6*sizeof(
            ↪ device_node_t)*available_samples);

        std::vector¡int> random_IDs(available_samples);
        std::iota(random_IDs.begin(), random_IDs.end(), 0);
        std::shuffle(random_IDs.begin(), random_IDs.end(), std::mt19937{42});
        std::vector¡int> id_subset(random_IDs.begin(), random_IDs.begin()+sample_size);
        for (int l = 0; l ¡ N_runs; l++){
            cuda_io::IsomerQueue isomer_q(N);
            cuda_io::IsomerQueue out_queue(N);
            out_queue.resize(2*sample_size);
            isomer_q.resize(2*sample_size);
            IsomerBatch batch0(N,M_b,DEVICE_BUFFER);
            for (int i = 0; i ¡ sample_size; ++i){
                for (size_t j = 0; j ¡ Nf; j++){
                    G.neighbours[j].clear();
                    for (size_t k = 0; k ¡ 6; k++) {
                        auto u = input_buffer[id_subset[i]*Nf*6 + j*6 +k];
                        if(u != UINT16_MAX) G.neighbours[j].push_back(u);
```

```
                }
            }
                auto Tstart = high_resolution_clock::now();
            G.update();
            PlanarGraph pG = G.dual_graph();
            pG.layout2d  = pG.tutte_layout();
                auto T3 = high_resolution_clock::now(); T_seq[l] += T3 − Tstart;
            Polyhedron P(pG);
                auto T4 = high_resolution_clock::now(); T_io[l] += T4 − T3;
            P.points     = P.zero_order_geometry();
                auto T5 = high_resolution_clock::now(); T_seq[l] += T5 − T4;
            isomer_q.insert(P,i,LaunchCtx(),LaunchPolicy::SYNC);
                auto T6 = high_resolution_clock::now(); T_io[l] += T6 − T5;
        }
        while (out_queue.get_size() ¡ sample_size){
        auto T0 = high_resolution_clock::now();
            isomer_q.refill_batch(batch0);
        auto T1 = high_resolution_clock::now(); T_io[l] += T1 − T0;
            gpu_kernels::isomerspace_forcefield::optimize¡BUSTER>(batch0,N*0.5,N*5);
        auto T2 = high_resolution_clock::now(); T_par[l] += T2 − T1;
            out_queue.push(batch0);
        auto T3 = high_resolution_clock::now(); T_io[l] += T3 − T2;
        }

    }
    using namespace cuda_io;
    auto total = (float)(mean(T_io)/1ns + mean(T_par)/1ns + mean(T_seq)/1ns);
    std::cout ¡¡ std::fixed ¡¡ std::setprecision(2) ¡¡ N ¡¡ ", "¡¡ sample_size ¡¡ ", " ¡¡
        ↪ (mean(T_seq)/1ns)/total*100. ¡¡ "%, " ¡¡ (mean(T_par)/1ns)/total*100. ¡¡ "%, "
        ↪ ¡¡ (mean(T_io)/1ns)/total*100. ¡¡ "%, " ¡¡ (float)(mean(T_io)/1us+mean(T_par)/1
        ↪ us+mean(T_seq)/1us)/sample_size ¡¡ "us/isomer\n";
    out_file ¡¡ N ¡¡ ", "¡¡ sample_size ¡¡ ", " ¡¡ mean(T_seq)/1ns ¡¡ ", " ¡¡ mean(T_par)
        ↪ /1ns ¡¡ ", " ¡¡ mean(T_io)/1ns¡¡ "\n";
    out_file_std ¡¡ N ¡¡ ", "¡¡ sample_size ¡¡ ", " ¡¡ sdev(T_seq)/1ns ¡¡ ", " ¡¡ sdev(
        ↪ T_par)/1ns ¡¡ ", " ¡¡ sdev(T_io)/1ns¡¡ "\n";
   }
  LaunchCtx::clear_allocations();
}
```

Listing C.3: *Benchmark Pipeline V2 Script*

```
#include "fullerenes/gpu/isomer_queue.hh"
#include "fullerenes/gpu/cuda_io.hh"
#include "fullerenes/gpu/kernels.hh"

const std::unordered_map¡size_t,size_t> num_fullerenes =
    ↪ {{20,1},{22,0},{24,1},{26,1},{28,2},{30,3},{32,6},{34,6},{36,15},{38,17},{40,40},{42,45},{44,89},{46,116},{48,1
    ↪ ;
using namespace chrono;
using namespace chrono_literals;

int main(int argc, char** argv){
    const size_t N_start                = argc>1 ? strtol(argv[1],0,0) : 20;
        ↪                     // Argument 1: Number of vertices N
    const size_t N_limit                = argc>2 ? strtol(argv[2],0,0) : 200;
        ↪                    // Argument 1: Number of vertices N

    auto N_runs = 10;
    ofstream out_file("IsomerspaceOpt_V2_" + to_string(N_limit) + ".txt");
    ofstream out_file_std("IsomerspaceOpt_V2_STD_" + to_string(N_limit) + ".txt");
    for (size_t N = N_start; N ¡ N_limit+1; N+=2)
    {
        if(N == 22) continue;
        auto Nf = N/2 + 2;
        auto sample_size = min(gpu_kernels::isomerspace_forcefield::optimal_batch_size(N,0)*
            ↪ 4,(int)num_fullerenes.find(N)->second);
        std::queue¡std::tuple¡Polyhedron,size_t,IsomerStatus>> poly_queue;

        //Queue needs to allocate memory the first time something is pushed, we don't want to
            ↪ measure this,
        //So we make sure the queue preallocates some memory.
        for (size_t i = 0; i ¡ sample_size; i++){
            poly_queue.push({Polyhedron(N),i,IsomerStatus::FAILED});
        }
        while (!poly_queue.empty()){
            poly_queue.pop();
        }

        FullereneDual G;
        G.neighbours = neighbours_t(Nf, std::vector¡node_t>(6));
        G.N = Nf;

        bool more_to_generate = true;

        auto T0 = high_resolution_clock::now();
        auto

            T_seq     = std::vector¡std::chrono::nanoseconds>(N_runs),
            T_par     = std::vector¡std::chrono::nanoseconds>(N_runs),
            T_io      = std::vector¡std::chrono::nanoseconds>(N_runs);

        auto path = "isomerspace_samples/dual_layout_" + to_string(N) + "_seed_42";
        ifstream isomer_sample(path,std::ios::binary);
        auto fsize = std::filesystem::file_size(path);
        std::vector¡device_node_t> input_buffer(fsize/sizeof(device_node_t));
        auto available_samples = fsize / (Nf*6*sizeof(device_node_t));
        isomer_sample.read(reinterpret_cast¡char*>(input_buffer.data()), Nf*6*sizeof(
            ↪ device_node_t)*available_samples);

        std::vector¡int> random_IDs(available_samples);
        std::iota(random_IDs.begin(), random_IDs.end(), 0);
        std::shuffle(random_IDs.begin(), random_IDs.end(), std::mt19937{42});
        std::vector¡int> id_subset(random_IDs.begin(), random_IDs.begin()+sample_size);
        for(int l = 0; l ¡ N_runs; l++){
        cuda_io::IsomerQueue Q0(N);
        cuda_io::IsomerQueue Q1(N);
        cuda_io::IsomerQueue Q2(N);
```

```
        IsomerBatch batch0(N,sample_size,DEVICE_BUFFER);
        IsomerBatch batch1(N,sample_size,DEVICE_BUFFER);
        IsomerBatch h_batch(N,sample_size,HOST_BUFFER);
        //Pre allocate the device queue such that it doesn't happen during benchmarking
        Q0.resize(2*sample_size);
        Q1.resize(2*sample_size);
        Q2.resize(2*sample_size);
        for (int i = 0; i ¡ sample_size; ++i){
            for (size_t j = 0; j ¡ Nf; j++){
                G.neighbours[j].clear();
                for (size_t k = 0; k ¡ 6; k++) {
                    auto u = input_buffer[id_subset[i]*Nf*6 + j*6 +k];
                    if(u != UINT16_MAX) G.neighbours[j].push_back(u);
                }
            }
                auto T1 = high_resolution_clock::now();
            G.update();
            PlanarGraph pG = G.dual_graph();
                auto T2 = high_resolution_clock::now(); T_seq[l] += (T2 - T1);
            Q0.insert(pG,i,LaunchCtx(),LaunchPolicy::SYNC, false);
                auto T3 = high_resolution_clock::now(); T_io[l] += (T3 - T2);
        }

        auto T1 = high_resolution_clock::now();
            Q0.refill_batch(batch0);
            gpu_kernels::isomerspace_tutte::tutte_layout(batch0);
        auto T2 = high_resolution_clock::now(); T_par[l] += (T2 - T1);
            cuda_io::copy(h_batch,batch0);
            cuda_io::output_to_queue(poly_queue,h_batch);
        auto T3 = high_resolution_clock::now(); T_io[l] += (T3 - T2);

        while (!poly_queue.empty())
        {
            auto wT1 = high_resolution_clock::now();
                auto [P, ID, status] = poly_queue.front();
            auto wT2 = high_resolution_clock::now(); T_io[l] += (wT2 - wT1);
                P.points = P.zero_order_geometry();
            auto wT3 = high_resolution_clock::now(); T_seq[l] += (wT3 - wT2);
                Q1.insert(P,ID);
            auto wT4 = high_resolution_clock::now(); T_io[l] += (wT4 - wT3);
            poly_queue.pop();
        }

        while (Q2.get_size() ¡ sample_size){
            auto T0 = high_resolution_clock::now();
                Q1.refill_batch(batch1);
            auto T1 = high_resolution_clock::now(); T_io[l] += T1 - T0;
                gpu_kernels::isomerspace_forcefield::optimize¡BUSTER>(batch1,N*0.5,N*5);
            auto T2 = high_resolution_clock::now(); T_par[l] += T2 - T1;
                Q2.push(batch1);
            auto T3 = high_resolution_clock::now(); T_io[l] += T3 - T2;
        }
        }
        using namespace cuda_io;
        auto total = (float)(mean(T_io)/1ns + mean(T_par)/1ns + mean(T_seq)/1ns);
        std::cout ¡¡ std::fixed ¡¡ std::setprecision(2) ¡¡ N ¡¡ ",␣"¡¡ sample_size ¡¡ ",␣" ¡¡
            ↪ (mean(T_seq)/1ns)/total*100. ¡¡ "%,␣" ¡¡ (mean(T_par)/1ns)/total*100. ¡¡ "%,␣"
            ↪ ¡¡ (mean(T_io)/1ns)/total*100. ¡¡ "%,␣" ¡¡ (float)(mean(T_io)/1us+mean(T_par)/1
            ↪ us+mean(T_seq)/1us)/sample_size ¡¡ "us/isomer\n";
        out_file ¡¡ N ¡¡ ",␣"¡¡ sample_size ¡¡ ",␣" ¡¡ mean(T_seq)/1ns ¡¡ ",␣" ¡¡ mean(T_par)
            ↪ /1ns ¡¡ ",␣" ¡¡ mean(T_io)/1ns ¡¡ "\n";
        out_file_std ¡¡ N ¡¡ ",␣"¡¡ sample_size ¡¡ ",␣" ¡¡ sdev(T_seq)/1ns ¡¡ ",␣" ¡¡ sdev(
            ↪ T_par)/1ns ¡¡ ",␣" ¡¡ sdev(T_io)/1ns ¡¡ "\n";
    }
    LaunchCtx::clear_allocations();
}
```

Listing C.4: *Benchmark Pipeline V3 Script*

```
#include "fullerenes/gpu/isomer_queue.hh"
#include "fullerenes/gpu/cuda_io.hh"
#include "fullerenes/gpu/kernels.hh"

const std::unordered_map¡size_t,size_t> num_fullerenes =
    ↪ {{20,1},{22,0},{24,1},{26,1},{28,2},{30,3},{32,6},{34,6},{36,15},{38,17},{40,40},{42,45},{44,89},{46,116},{48,1
    ↪ ;
using namespace chrono;
using namespace chrono_literals;

int main(int argc, char** argv){
    const size_t N_start                = argc>1 ? strtol(argv[1],0,0) : 20;
        ↪                    // Argument 1: Number of vertices N
    const size_t N_limit                = argc>2 ? strtol(argv[2],0,0) : 200;
        ↪                   // Argument 1: Number of vertices N
    auto N_runs = 10;
    ofstream out_file("IsomerspaceOpt_V3_" + to_string(N_limit) + ".txt");
    ofstream out_file_std("IsomerspaceOpt_V3_STD_" + to_string(N_limit) + ".txt");
    for (size_t N = N_start; N ¡ N_limit+1; N+=2)
    {
        if (N == 22) continue; //No isomers in isomerspace 22;
        auto sample_size = min(gpu_kernels::isomerspace_forcefield::optimal_batch_size(N,0)*
            ↪ 4,(int)num_fullerenes.find(N)->second);
        auto M_b = min(gpu_kernels::isomerspace_forcefield::optimal_batch_size(N,0),(int)
            ↪ num_fullerenes.find(N)->second);
        std::queue¡std::tuple¡Polyhedron, size_t, IsomerStatus>> poly_queue;
        FullereneDual G;

        bool more_to_generate = true;

        auto T0 = high_resolution_clock::now();
        auto

            T_seq     = std::vector¡std::chrono::nanoseconds>(N_runs),
            T_par     = std::vector¡std::chrono::nanoseconds>(N_runs),
            T_io      = std::vector¡std::chrono::nanoseconds>(N_runs);

        auto Nf = N / 2 + 2;
        auto path = "isomerspace_samples/dual_layout_" + to_string(N) + "_seed_42";
        ifstream isomer_sample(path,std::ios::binary);
        auto fsize = std::filesystem::file_size(path);
        std::vector¡device_node_t> input_buffer(fsize/sizeof(device_node_t));
        auto available_samples = fsize / (Nf*6*sizeof(device_node_t));
        isomer_sample.read(reinterpret_cast¡char*>(input_buffer.data()), Nf*6*sizeof(
            ↪ device_node_t)*available_samples);

        std::vector¡int> random_IDs(available_samples);
        std::iota(random_IDs.begin(), random_IDs.end(), 0);
        std::shuffle(random_IDs.begin(), random_IDs.end(), std::mt19937{42});
        std::vector¡int> id_subset(random_IDs.begin(), random_IDs.begin()+sample_size);

        G.neighbours = neighbours_t(Nf, std::vector¡node_t>(6));
        G.N = Nf;
        for (size_t l = 0; l ¡ N_runs; l++){
            IsomerBatch batch0(N,sample_size,DEVICE_BUFFER);
            IsomerBatch batch1(N,M_b,DEVICE_BUFFER);
            cuda_io::IsomerQueue Q0(N);
            cuda_io::IsomerQueue Q1(N);
            cuda_io::IsomerQueue Q2(N);
            //Pre allocate the device queue such that it doesn't happen during benchmarking
            Q0.resize(2*sample_size);
            Q1.resize(2*sample_size);
            Q2.resize(2*sample_size);
            for (int i = 0; i ¡ sample_size; ++i){
                for (size_t j = 0; j ¡ Nf; j++){
                    G.neighbours[j].clear();
```

```
                    for (size_t k = 0; k ¡ 6; k++) {
                        auto u = input_buffer[id_subset[i]*Nf*6 + j*6 +k];
                        if(u != UINT16_MAX) G.neighbours[j].push_back(u);
                    }
                }
                        auto T0 = high_resolution_clock::now();
                    G.update();
                    PlanarGraph pG = G.dual_graph();
                        auto T2 = high_resolution_clock::now(); T_seq[l] += (T2 - T0);
                    Q0.insert(pG,i,LaunchCtx(),LaunchPolicy::SYNC, false);
                        auto T3 = high_resolution_clock::now(); T_io[l] += (T3 - T2);
            }

            auto T1 = high_resolution_clock::now();
                Q0.refill_batch(batch0);
            auto T2 = high_resolution_clock::now(); T_io[l] += (T2 - T1);
                gpu_kernels::isomerspace_tutte::tutte_layout(batch0);
                gpu_kernels::isomerspace_X0::zero_order_geometry(batch0, 4.0);
            auto T3 = high_resolution_clock::now(); T_par[l] += (T3 - T2);
                Q1.insert(batch0);
            auto T4 = high_resolution_clock::now(); T_io[l] += (T4 - T3);

            while (Q2.get_size() ¡ sample_size)
            {
                auto T0 = high_resolution_clock::now();
                Q1.refill_batch(batch1);
                auto T1 = high_resolution_clock::now(); T_io[l] += (T1 - T0);
                gpu_kernels::isomerspace_forcefield::optimize¡BUSTER>(batch1,N*0.5,N*5);
                auto T2 = high_resolution_clock::now(); T_par[l] += (T2 - T1);
                Q2.push(batch1);
                auto T3 = high_resolution_clock::now(); T_io[l] += (T3 - T2);
            }
        }
        using namespace cuda_io;
        auto total = (float)(mean(T_io)/1ns + mean(T_par)/1ns + mean(T_seq)/1ns);
        std::cout ¡¡ std::fixed ¡¡ std::setprecision(2) ¡¡ N ¡¡ ",␣"¡¡ sample_size ¡¡ ",␣" ¡¡
            ↪ (mean(T_seq)/1ns)/total*100. ¡¡ "%,␣" ¡¡ (mean(T_par)/1ns)/total*100. ¡¡ "%,␣"
            ↪ ¡¡ (mean(T_io)/1ns)/total*100. ¡¡ "%,␣" ¡¡ (float)(mean(T_io)/1us+mean(T_par)/1
            ↪ us+mean(T_seq)/1us)/sample_size ¡¡ "us/isomer\n";
        out_file ¡¡ N ¡¡ ",␣"¡¡ sample_size ¡¡ ",␣" ¡¡ mean(T_seq)/1ns ¡¡ ",␣" ¡¡ mean(T_par)
            ↪ /1ns ¡¡ ",␣" ¡¡ mean(T_io)/1ns ¡¡ "\n";
        out_file_std ¡¡ N ¡¡ ",␣"¡¡ sample_size ¡¡ ",␣" ¡¡ sdev(T_seq)/1ns ¡¡ ",␣" ¡¡ sdev(
            ↪ T_par)/1ns ¡¡ ",␣" ¡¡ sdev(T_io)/1ns ¡¡ "\n";
    }
LaunchCtx::clear_allocations();
}
```

Listing C.5: *Benchmark Pipeline V4 Script*

```
#include "fullerenes/gpu/isomer_queue.hh"
#include "fullerenes/gpu/cuda_io.hh"
#include "fullerenes/gpu/kernels.hh"

const std::unordered_map<size_t,size_t> num_fullerenes =
    ↪ {{20,1},{22,0},{24,1},{26,1},{28,2},{30,3},{32,6},{34,6},{36,15},{38,17},{40,40},{42,45},{44,89},{46,116},{48,1
    ↪ ;
using namespace chrono;
using namespace chrono_literals;

int main(int argc, char** argv){
    const size_t N_start                = argc>1 ? strtol(argv[1],0,0) : 20;
        ↪                     // Argument 1: Number of vertices N
    const size_t N_limit                = argc>2 ? strtol(argv[2],0,0) : 200;
        ↪                    // Argument 1: Number of vertices N
    auto N_runs = 10;

    ofstream out_file("IsomerspaceOpt_V4_" + to_string(N_limit) + ".txt");
    ofstream out_file_std("IsomerspaceOpt_V4_STD_" + to_string(N_limit) + ".txt");
    for (size_t N = N_start; N < N_limit+1; N+=2)
    {
        if (N == 22) continue; //No isomers in isomerspace 22;
        auto sample_size = min(gpu_kernels::isomerspace_forcefield::optimal_batch_size(N,0)*
            ↪ 4,(int)num_fullerenes.find(N)->second);
        auto M_b = min(gpu_kernels::isomerspace_forcefield::optimal_batch_size(N,0),(int)
            ↪ num_fullerenes.find(N)->second);
        std::queue<std::tuple<Polyhedron,size_t,IsomerStatus>> poly_queue;

        bool more_to_generate = true;
        int N_f = N/2 + 2;
        auto T0 = high_resolution_clock::now();
        auto

            T_seq     = std::vector<std::chrono::nanoseconds>(N_runs, chrono::nanoseconds(1)),
            T_par     = std::vector<std::chrono::nanoseconds>(N_runs, chrono::nanoseconds(1)),
            T_io      = std::vector<std::chrono::nanoseconds>(N_runs, chrono::nanoseconds(1));

        Graph G;
        auto Nf = N/2 +2;
        std::string path = "isomerspace_samples/dual_layout_" + to_string(N) + "_seed_42";
        auto fsize = std::filesystem::file_size(path);
        auto n_samples = fsize / (Nf * 6 * sizeof(device_node_t));
        ifstream in_file(path,std::ios::binary);
        std::vector<device_node_t> dual_neighbours(n_samples * Nf * 6);
        in_file.read((char*)dual_neighbours.data(),n_samples * Nf * 6 * sizeof(device_node_t))
            ↪ ;

        std::vector<int> random_IDs(n_samples);
        std::iota(random_IDs.begin(), random_IDs.end(), 0);
        std::shuffle(random_IDs.begin(), random_IDs.end(), std::mt19937{42});
        std::vector<int> id_subset(random_IDs.begin(), random_IDs.begin()+sample_size);

        G.neighbours = neighbours_t(Nf, std::vector<node_t>(6));
        G.N = Nf;
        std::cout << "Isomerspace_:" << N << std::endl;
        for (size_t l = 0; l < N_runs; l++)
        {
            IsomerBatch batch0(N,sample_size,DEVICE_BUFFER);
            IsomerBatch batch1(N,M_b,DEVICE_BUFFER);
            cuda_io::IsomerQueue Q0(N,0);
            cuda_io::IsomerQueue Q1(N,0);
            cuda_io::IsomerQueue Q2(N,0);

            //Pre allocate the device queue such that it doesn't happen during benchmarking
            Q0.resize(2*sample_size);
            Q1.resize(2*sample_size);
```

```
            Q2.resize(2*sample_size);

            for (int i = 0; i ¡ sample_size; ++i){
                for (size_t j = 0; j ¡ Nf; j++){
                    G.neighbours[j].clear();
                    for (size_t k = 0; k ¡ 6; k++) {
                        auto u = dual_neighbours[id_subset[i]*Nf*6 + j*6 +k];
                        if(u != UINT16_MAX) G.neighbours[j].push_back(u);
                    }
                }
                    auto T2 = high_resolution_clock::now();
                Q0.insert(G,i);
                    auto T3 = high_resolution_clock::now(); T_io[l] += (T3 - T2);
            }

            auto T1 = high_resolution_clock::now();
                Q0.refill_batch(batch0);
            auto T2 = high_resolution_clock::now(); T_io[l] += (T2 - T1);
                gpu_kernels::isomerspace_dual::dualize(batch0);
                gpu_kernels::isomerspace_tutte::tutte_layout(batch0);
                gpu_kernels::isomerspace_X0::zero_order_geometry(batch0, 4.0);
            auto T3 = high_resolution_clock::now(); T_par[l] += (T3 - T2);
                Q1.insert(batch0);
            auto T4 = high_resolution_clock::now(); T_io[l] += (T4 - T3);
            while(Q2.get_size() ¡ sample_size){
                auto T1 = high_resolution_clock::now();
                    Q1.refill_batch(batch1);
                auto T2 = high_resolution_clock::now(); T_io[l] += (T2 - T1);
                    gpu_kernels::isomerspace_forcefield::optimize¡BUSTER>(batch1,N*0.5,N*5);
                auto T3 = high_resolution_clock::now(); T_par[l] += (T3 - T2);
                    Q2.push(batch1);
                auto T4 = high_resolution_clock::now(); T_io[l] += (T4 - T3);
            }
        }
        using namespace cuda_io;
        auto total = (float)(mean(T_io)/1ns + mean(T_par)/1ns + mean(T_seq)/1ns);
        std::cout ¡¡ std::fixed ¡¡ std::setprecision(2) ¡¡ N ¡¡ ",␣"¡¡ sample_size ¡¡ ",␣" ¡¡
            ↪ (mean(T_seq)/1ns)/total*100. ¡¡ "%,␣" ¡¡ (mean(T_par)/1ns)/total*100. ¡¡ "%,␣"
            ↪ ¡¡ (mean(T_io)/1ns)/total*100. ¡¡ "%,␣" ¡¡ (float)(mean(T_io)/1us+mean(T_par)/1
            ↪ us+mean(T_seq)/1us)/sample_size ¡¡ "us/isomer\n";
        out_file ¡¡ N ¡¡ ",␣"¡¡ sample_size ¡¡ ",␣" ¡¡ mean(T_seq) /1ns ¡¡ ",␣" ¡¡ mean(T_par)
            ↪ /1ns ¡¡ ",␣" ¡¡ mean(T_io)/1ns ¡¡ "\n";
        out_file_std ¡¡ N ¡¡ ",␣"¡¡ sample_size ¡¡ ",␣" ¡¡ sdev(T_seq) /1ns ¡¡ ",␣" ¡¡ sdev(
            ↪ T_par)/1ns ¡¡ ",␣" ¡¡ sdev(T_io)/1ns ¡¡ "\n";
    }
    LaunchCtx::clear_allocations();
}
```

Listing C.6: *Benchmark Pipeline V5 Script*

```
#include <future>
#include <thread>
#include "fullerenes/gpu/isomer_queue.hh"
#include "fullerenes/gpu/cuda_io.hh"
#include "fullerenes/gpu/kernels.hh"

const std::unordered_map<size_t,size_t> num_fullerenes =
    ↪ {{20,1},{22,0},{24,1},{26,1},{28,2},{30,3},{32,6},{34,6},{36,15},{38,17},{40,40},{42,45},{44,89},{46,116},{48,1
    ↪ ;
using namespace chrono;
using namespace chrono_literals;
using namespace gpu_kernels;
int main(int argc, char** argv){
    const size_t N_start                = argc > 1 ? strtol(argv[1],0,0) : 20;     // Argument
        ↪  1: Number of vertices N
    const size_t N_limit                = argc > 2 ? strtol(argv[2],0,0) : 200;     //
        ↪ Argument 1: Number of vertices N
    const size_t N_runs                 = argc > 3 ? strtol(argv[3],0,0) : 10;

    ofstream out_file("IsomerspaceOpt_V5_" + to_string(N_limit) + ".txt");
    ofstream out_file_std("IsomerspaceOpt_V5_STD_" + to_string(N_limit) + ".txt");


    for (size_t N = N_start; N < N_limit+1; N+=2)
    {
        if (N == 22) continue; //No isomers in isomerspace 22;
        auto n_fullerenes = (int)num_fullerenes.find(N)->second;
        auto batch_size = min(isomerspace_forcefield::optimal_batch_size(N), n_fullerenes);
        auto sample_size = min(isomerspace_forcefield::optimal_batch_size(N)*4, n_fullerenes);

        bool more_to_generate = true;
        int N_f = N/2 + 2;
        auto I = 0;
        auto T0 = high_resolution_clock::now();
        auto

            T_seq     = std::vector<std::chrono::nanoseconds>(N_runs, chrono::nanoseconds(1)),
            T_par     = std::vector<std::chrono::nanoseconds>(N_runs, chrono::nanoseconds(1)),
            T_io      = std::vector<std::chrono::nanoseconds>(N_runs, chrono::nanoseconds(1));

        Graph G;
        auto Nf = N/2 +2;
        std::string path = "isomerspace_samples/dual_layout_" + to_string(N) + "_seed_42";
        auto fsize = std::filesystem::file_size(path);
        auto n_samples = fsize / (Nf * 6 * sizeof(device_node_t));
        ifstream in_file(path,std::ios::binary);
        std::vector<device_node_t> dual_neighbours(n_samples * Nf * 6);
        in_file.read((char*)dual_neighbours.data(),n_samples * Nf * 6 * sizeof(device_node_t))
            ↪ ;

        std::vector<int> random_IDs(n_samples);
        std::iota(random_IDs.begin(), random_IDs.end(), 0);
        std::shuffle(random_IDs.begin(), random_IDs.end(), std::mt19937{42});
        std::vector<int> id_subset(random_IDs.begin(), random_IDs.begin()+sample_size);

        for (size_t l = 0; l < N_runs; ++l)
        {
        IsomerBatch B0(N,batch_size,DEVICE_BUFFER);
        IsomerBatch B1(N,batch_size,DEVICE_BUFFER);
        cuda_io::IsomerQueue Q0(N);
        cuda_io::IsomerQueue Q1(N);
        cuda_io::IsomerQueue Q2(N);
        //Pre allocate the device queue such that it doesn't happen during benchmarking
        Q0.resize(2*sample_size);
        Q1.resize(2*sample_size);
        Q2.resize(2*sample_size);
```

```
    G.neighbours = neighbours_t(Nf, std::vector¡node_t>(6));
    G.N = Nf;
    LaunchCtx insert_ctx = LaunchCtx(0);
    LaunchCtx device0 = LaunchCtx(0);
    auto generate_isomers = [&](){
        for (int i = 0; i ¡ batch_size; ++i){
            for (size_t j = 0; j ¡ Nf; j++){
                G.neighbours[j].clear();
                for (size_t k = 0; k ¡ 6; k++) {
                    auto u = dual_neighbours[id_subset[i]*Nf*6 + j*6 +k];
                    if(u != UINT16_MAX) G.neighbours[j].push_back(u);
                }
            }
            I++;
            Q0.insert(G,i, insert_ctx, LaunchPolicy::ASYNC);
        }
        insert_ctx.wait();
    };

    generate_isomers();
    while(Q2.get_size() ¡ sample_size){
        Q0.refill_batch(B0);
        auto generate_handle = std::async(std::launch::async, generate_isomers);
        auto T1 = high_resolution_clock::now();
            isomerspace_dual::dualize(B0,device0,LaunchPolicy::ASYNC);
            isomerspace_tutte::tutte_layout(B0,10000000,device0,LaunchPolicy::ASYNC);
            isomerspace_X0::zero_order_geometry(B0, 4.0, device0, LaunchPolicy::ASYNC);
            device0.wait();
        auto T2 = high_resolution_clock::now(); T_par[l] += ( T2 - T1);
            Q1.insert(B0, device0, LaunchPolicy::ASYNC);
            device0.wait();
        auto T3 = high_resolution_clock::now(); T_io[l] += (T3 - T2);
        while(Q1.get_size() >= B1.capacity()){
            auto T1 = high_resolution_clock::now();
                Q1.refill_batch(B1);
            auto T2 = high_resolution_clock::now(); T_io[l] += (T2 - T1);
                isomerspace_forcefield::optimize¡BUSTER>(B1,N*0.5,N*5);
            auto T3 = high_resolution_clock::now(); T_par[l] += (T3 - T2);
                Q2.push(B1, device0);
            auto T4 = high_resolution_clock::now(); T_io[l] += (T4 - T3);
        }
            generate_handle.wait();
            while (I >= sample_size && Q2.get_size() ¡ sample_size && Q0.get_size() == 0)
                ↪ {
                auto T1 = high_resolution_clock::now();
                    Q1.refill_batch(B1);
                auto T2 = high_resolution_clock::now(); T_io[l] += (T2 - T1);
                    isomerspace_forcefield::optimize¡BUSTER>(B1,N*0.5,N*5);
                auto T3 = high_resolution_clock::now(); T_par[l] += (T3 - T2);
                    Q2.push(B1);
                auto T4 = high_resolution_clock::now(); T_io[l] += (T4 - T3);
            }
    }

    }
    using namespace cuda_io;
    auto total = (float)(mean(T_io)/1ns + mean(T_par)/1ns + mean(T_seq)/1ns);
    std::cout ¡¡ std::fixed ¡¡ std::setprecision(2) ¡¡ N ¡¡ ",␣"¡¡ sample_size ¡¡ ",␣" ¡¡
        ↪ (mean(T_seq)/1ns)/total*100. ¡¡ "%,␣" ¡¡ (mean(T_par)/1ns)/total*100. ¡¡ "%,␣"
        ↪ ¡¡ (mean(T_io)/1ns)/total*100. ¡¡ "%,␣" ¡¡ (float)(mean(T_io)/1us+mean(T_par)/1
        ↪ us+mean(T_seq)/1us)/sample_size ¡¡ "us/isomer\n";
    out_file ¡¡ N ¡¡ ",␣"¡¡ sample_size ¡¡ ",␣" ¡¡ mean(T_seq) /1ns ¡¡ ",␣" ¡¡ mean(T_par)
        ↪ /1ns ¡¡ ",␣" ¡¡ mean(T_io)/1ns ¡¡ "\n";
    out_file_std ¡¡ N ¡¡ ",␣"¡¡ sample_size ¡¡ ",␣" ¡¡ sdev(T_seq) /1ns ¡¡ ",␣" ¡¡ sdev(
        ↪ T_par)/1ns ¡¡ ",␣" ¡¡ sdev(T_io)/1ns ¡¡ "\n";
 }
LaunchCtx::clear_allocations();
```

```
}
```

Listing C.7: *Benchmark Pipeline V6 Script*

```
#include ¡future>
#include ¡thread>
#include "fullerenes/gpu/isomer_queue.hh"
#include "fullerenes/gpu/cuda_io.hh"
#include "fullerenes/gpu/kernels.hh"

const std::unordered_map¡size_t,size_t> num_fullerenes =
    ↪ {{20,1},{22,0},{24,1},{26,1},{28,2},{30,3},{32,6},{34,6},{36,15},{38,17},{40,40},{42,45},{44,89},{46,116}
    ↪ ;
using namespace chrono;
using namespace chrono_literals;
using namespace gpu_kernels;
int main(int argc, char** argv){
    const size_t N_start            = argc > 1 ? strtol(argv[1],0,0) : 20;     // Argument
        ↪  1: Number of vertices N
    const size_t N_limit            = argc > 2 ? strtol(argv[2],0,0) : N_start;     //
        ↪ Argument 1: Number of vertices N
    auto N_runs = 10;

    ofstream out_file("IsomerspaceOpt_V6_" + to_string(N_limit) + ".txt");
    ofstream out_file_std("IsomerspaceOpt_V6_STD_" + to_string(N_limit) + ".txt");
    for (size_t N = N_start; N ¡ N_limit+1; N+=2)
    {
        if (N == 22) continue; //No isomers in isomerspace 22;
        auto batch_size = isomerspace_forcefield::optimal_batch_size(N);
        auto n_fullerenes = (int)num_fullerenes.find(N)->second;
        auto sample_size = min(batch_size*1,n_fullerenes);
        if (n_fullerenes ¡ batch_size*2){
            sample_size = max(n_fullerenes/2,1);
        }else if(n_fullerenes >= batch_size*8){
            sample_size = batch_size*4;
        }else if (n_fullerenes >= batch_size*6){
            sample_size = batch_size*3;
        }else if (n_fullerenes >= batch_size*4){
            sample_size = batch_size*2;
        } else if (n_fullerenes >= batch_size*2){
            sample_size = batch_size;
        }

        std::cout ¡¡ N ¡¡ endl;


        bool more_to_generate = true;
        int N_f = N/2 + 2;
        auto T0 = high_resolution_clock::now();
        auto

            T_seq     = std::vector¡std::chrono::nanoseconds>(N_runs, chrono::nanoseconds(1)),
            T_par     = std::vector¡std::chrono::nanoseconds>(N_runs, chrono::nanoseconds(1)),
            T_io      = std::vector¡std::chrono::nanoseconds>(N_runs, chrono::nanoseconds(1));

        Graph G;
        auto Nf = N/2 +2;
        std::string path = "isomerspace_samples/dual_layout_" + to_string(N) + "_seed_42";
        auto fsize = std::filesystem::file_size(path);
        auto n_samples = fsize / (Nf * 6 * sizeof(device_node_t));
        ifstream in_file(path,std::ios::binary);
        std::vector¡device_node_t> dual_neighbours(n_samples * Nf * 6);
        in_file.read((char*)dual_neighbours.data(),n_samples * Nf * 6 * sizeof(device_node_t))
            ↪ ;

        std::vector¡int> random_IDs(n_samples);
        std::iota(random_IDs.begin(), random_IDs.end(), 0);
        std::shuffle(random_IDs.begin(), random_IDs.end(), std::mt19937{42});
        std::vector¡int> id_subset(random_IDs.begin(), random_IDs.begin()+sample_size);
```

```
G.neighbours = neighbours_t(Nf, std::vector¡node_t>(6));
G.N = Nf;


for (size_t l = 0; l ¡ N_runs; l++)
{
IsomerBatch B0(N,sample_size,DEVICE_BUFFER, 0);
IsomerBatch B1(N,sample_size,DEVICE_BUFFER, 1);
cuda_io::IsomerQueue In_Q0(N, 0);
cuda_io::IsomerQueue In_Q1(N, 1);
cuda_io::IsomerQueue Out_Q0(N, 0);
cuda_io::IsomerQueue Out_Q1(N, 1);
LaunchCtx insert0_ctx = LaunchCtx(0);
LaunchCtx insert1_ctx = LaunchCtx(1);
LaunchCtx device0 = LaunchCtx(0);
LaunchCtx device1 = LaunchCtx(1);

//Pre allocate the device queue such that it doesn't happen during benchmarking
In_Q0.resize(2*sample_size, insert0_ctx);
In_Q1.resize(2*sample_size, insert1_ctx);
Out_Q0.resize(2*sample_size, insert0_ctx);
Out_Q1.resize(2*sample_size, insert1_ctx);


auto generate_isomers = [&](){
    for (int i = 0; i ¡ sample_size; ++i){
        for (size_t j = 0; j ¡ Nf; j++){
            G.neighbours[j].clear();
            for (size_t k = 0; k ¡ 6; k++) {
                auto u = dual_neighbours[id_subset[i]*Nf*6 + j*6 +k];
                if(u != UINT16_MAX) G.neighbours[j].push_back(u);
            }
        }
        In_Q0.insert(G,i, insert0_ctx, LaunchPolicy::ASYNC);
        In_Q1.insert(G,i,insert1_ctx, LaunchPolicy::ASYNC);
    }
    insert0_ctx.wait(); insert1_ctx.wait();
};

//Produce and insert the first batch
generate_isomers();
In_Q0.refill_batch(B0, insert0_ctx, LaunchPolicy::SYNC);
In_Q1.refill_batch(B1, insert1_ctx, LaunchPolicy::SYNC);

    //Generate new isomers while processing the previous batch
    auto generate_handle = std::async(std::launch::async, generate_isomers);
    auto T2 = high_resolution_clock::now();
        //Main processing
        isomerspace_dual::dualise(B0, device0, LaunchPolicy::ASYNC);
        isomerspace_dual::dualise(B1, device1, LaunchPolicy::ASYNC);
        isomerspace_tutte::tutte_layout(B0, 10000000, device0, LaunchPolicy::ASYNC);
        isomerspace_tutte::tutte_layout(B1, 10000000, device1, LaunchPolicy::ASYNC);
        isomerspace_X0::zero_order_geometry(B0, 4.0, device0, LaunchPolicy::ASYNC);
        isomerspace_X0::zero_order_geometry(B1, 4.0, device1, LaunchPolicy::ASYNC);
        isomerspace_forcefield::optimise¡PEDERSEN>(B0,N*5,N*5, device0, LaunchPolicy::
            ↪ ASYNC);
        isomerspace_forcefield::optimise¡PEDERSEN>(B1,N*5,N*5, device1, LaunchPolicy::
            ↪ ASYNC);
        //Output finished isomers
        Out_Q0.push(B0,device0,LaunchPolicy::ASYNC);
        Out_Q1.push(B1,device1,LaunchPolicy::ASYNC);
        device0.wait(); device1.wait();
    auto T3 = high_resolution_clock::now(); T_par[l] += (T3 - T2);
        //Wait for generation to finish, if this process is faster than GPU operations
            ↪ , the call returns immediately.
        generate_handle.wait();
        //Refill batches with new isomers
```

```
                In_Q0.refill_batch(B0, device0, LaunchPolicy::ASYNC);
                In_Q1.refill_batch(B1, device1, LaunchPolicy::ASYNC);
                device0.wait(); device1.wait();
            auto T4 = high_resolution_clock::now(); T_io[l] += (T4 - T3);

        }
        using namespace cuda_io;
        out_file ¡¡ N ¡¡ ",␣"¡¡ min(sample_size*2, n_fullerenes) ¡¡ ",␣" ¡¡ mean(T_seq)/1ns ¡
            ↪ ¡ ",␣" ¡¡ mean(T_par)/1ns ¡¡ ",␣" ¡¡ mean(T_io)/1ns ¡¡ "\n";
        out_file_std ¡¡ N ¡¡ ",␣"¡¡ min(sample_size*2, n_fullerenes) ¡¡ ",␣" ¡¡ sdev(T_seq)/1
            ↪ ns ¡¡ ",␣" ¡¡ sdev(T_par)/1ns ¡¡ ",␣" ¡¡ sdev(T_io)/1ns ¡¡ "\n";
    }
    LaunchCtx::clear_allocations();
}
```

Listing C.8: *Benchmark Pipeline V7 Script*

```
#include ¡numeric>
#include ¡future>
#include ¡random>
const std::unordered_map¡size_t,size_t> num_fullerenes =
    ↪ {{20,1},{22,0},{24,1},{26,1},{28,2},{30,3},{32,6},{34,6},{36,15},{38,17},{40,40},{42,45},{44,89},{46,116},{48,1
    ↪ ;
using namespace gpu_kernels;
using namespace cuda_io;
#define SYNC LaunchPolicy::SYNC
#define ASYNC LaunchPolicy::ASYNC

int main(int ac, char** argv){
    int N_start             = ac > 1 ? strtol(argv[1],0,0) : 20;     // Argument 1: Number
        ↪ of vertices N
    int N_end               = ac > 2 ? strtol(argv[2],0,0) : N_start;     // Argument 1:
        ↪ Number of vertices N
    auto N_runs = 3;

    ofstream out_file("IsomerspaceOpt_V7_" + to_string(N_end) + ".txt");
    ofstream out_file_std("IsomerspaceOpt_V7_STD_" + to_string(N_end) + ".txt");
    for(int N = N_start; N ¡ N_end + 1;  N+=2){
        if(N == 22) continue;
        auto n_fullerenes = num_fullerenes.find(N)->second;
        Graph G;
        auto Nf = N/2 +2;
        G.neighbours = neighbours_t(Nf, std::vector¡node_t>(6));
        G.N = Nf;
        auto bucky = BuckyGen::start(N, false, false);

        std::string path = "isomerspace_samples/dual_layout_" + to_string(N) + "_seed_42";
        auto fsize = std::filesystem::file_size(path);
        auto n_samples = fsize / (Nf * 6 * sizeof(device_node_t));
        ifstream in_file(path,std::ios::binary);
        std::vector¡device_node_t> dual_neighbours(n_samples * Nf * 6);
        in_file.read((char*)dual_neighbours.data(),n_samples * Nf * 6 * sizeof(device_node_t))
            ↪ ;
        auto optimal_batch_size = isomerspace_forcefield::optimal_batch_size(N);
        auto sample_size = min(optimal_batch_size,(int)n_samples);
        if (n_fullerenes ¡ optimal_batch_size*2){
            sample_size = max((int)n_fullerenes/2,1);
        }else if(n_fullerenes >= optimal_batch_size*8){
            sample_size = optimal_batch_size*4;
        }else if (n_fullerenes >= optimal_batch_size*6){
            sample_size = optimal_batch_size*3;
        }else if (n_fullerenes >= optimal_batch_size*4){
            sample_size = optimal_batch_size*2;
        } else if (n_fullerenes >= optimal_batch_size*2){
            sample_size = optimal_batch_size;
        }

        std::vector¡int> random_IDs(n_samples);
        std::iota(random_IDs.begin(), random_IDs.end(), 0);
        std::shuffle(random_IDs.begin(), random_IDs.end(), std::mt19937{42});
        auto id_range_end = min((int)n_samples, sample_size);
        std::vector¡int> id_subset(random_IDs.begin(), random_IDs.begin()+n_samples);
        auto
            T_ends    = std::vector¡std::chrono::nanoseconds>(N_runs, chrono::nanoseconds(1)),
            T_par     = std::vector¡std::chrono::nanoseconds>(N_runs, chrono::nanoseconds(1)),
            T_io      = std::vector¡std::chrono::nanoseconds>(N_runs, chrono::nanoseconds(1));

        auto finished_fullerenes = 0;
        //ACTUAL PIPELINE CODE
        for(int i = 0; i ¡ N_runs; i++){
        finished_fullerenes = 0;
        LaunchCtx insert0_ctx(0);
        LaunchCtx insert1_ctx(1);
```

```
LaunchCtx device0_ctx(0);
LaunchCtx device1_ctx(1);

IsomerQueue input0_queue(N, 0);
IsomerQueue input1_queue(N, 1);
IsomerQueue opt0_queue(N, 0);
IsomerQueue opt1_queue(N, 1);
IsomerQueue output0_queue(N,0);
IsomerQueue output1_queue(N,1);
input0_queue.resize(sample_size*2);
input1_queue.resize(sample_size*2);
output0_queue.resize(sample_size);
output1_queue.resize(sample_size);
opt0_queue.resize(n_samples*2);
opt1_queue.resize(n_samples*2);

IsomerBatch input0(N, sample_size*2, DEVICE_BUFFER, 0);
IsomerBatch input1(N, sample_size*2, DEVICE_BUFFER, 1);
IsomerBatch opt0(N, sample_size, DEVICE_BUFFER, 0);
IsomerBatch opt1(N, sample_size, DEVICE_BUFFER, 1);

int I_async = 0;
auto generate_isomers = [&](int M){
    if(I_async == min(n_fullerenes,n_samples*10)) return false;
    for (int i = 0; i < M; i++){
        if (I_async < min(n_fullerenes,n_samples*10)){
            for (size_t j = 0; j < Nf; j++){
                G.neighbours[j].clear();
                for (size_t k = 0; k < 6; k++) {
                    auto u = dual_neighbours[id_subset[I_async%n_samples]*Nf*6 + j*6 +
                        ↪ k];
                    if(u != UINT16_MAX) G.neighbours[j].push_back(u);
                }
            }
            input0_queue.insert(G,I_async, insert0_ctx, ASYNC);
            input1_queue.insert(G,I_async, insert1_ctx, ASYNC);
            I_async++;
        }
    }

    input0_queue.refill_batch(input0, insert0_ctx, ASYNC);
    input1_queue.refill_batch(input1, insert1_ctx, ASYNC);
    isomerspace_dual::dualize(input0, insert0_ctx, ASYNC);
    isomerspace_dual::dualize(input1, insert1_ctx, ASYNC);
    isomerspace_tutte::tutte_layout(input0, 1000000, insert0_ctx, ASYNC);
    isomerspace_tutte::tutte_layout(input1, 1000000, insert1_ctx, ASYNC);
    isomerspace_X0::zero_order_geometry(input0, 4.0, insert0_ctx, ASYNC);
    isomerspace_X0::zero_order_geometry(input1, 4.0, insert1_ctx, ASYNC);
    insert0_ctx.wait(); insert1_ctx.wait();

    return I_async < min(size_t(ceil(n_fullerenes/2)),n_samples*10);
};
auto T1 = chrono::high_resolution_clock::now();
generate_isomers(sample_size*2);

opt0_queue.insert(input0, device0_ctx, ASYNC);
opt1_queue.insert(input1, device1_ctx, ASYNC);
opt0_queue.refill_batch(opt0, device0_ctx, ASYNC);
opt1_queue.refill_batch(opt1, device1_ctx, ASYNC);
device0_ctx.wait(); device1_ctx.wait();
T_ends[i] += T1 - chrono::high_resolution_clock::now();
bool more_to_do = true;
bool more_to_generate = false;
auto step = max(1, (int)N/2);

while (more_to_do){
    bool optimize_more = true;
```

```
        auto generate_handle = std::async(std::launch::async,generate_isomers, opt0.
            ↪ isomer_capacity*2);
        while(optimize_more){
            auto T2 = chrono::high_resolution_clock::now();
            isomerspace_forcefield::optimize¡BUSTER>(opt0,step, N*5, device0_ctx, ASYNC);
            isomerspace_forcefield::optimize¡BUSTER>(opt1,step, N*5, device1_ctx, ASYNC);
            output0_queue.push(opt0, device0_ctx, ASYNC);
            output1_queue.push(opt1, device1_ctx, ASYNC);
            opt0_queue.refill_batch(opt0, device0_ctx, ASYNC);
            opt1_queue.refill_batch(opt1, device1_ctx, ASYNC);
            device0_ctx.wait(); device1_ctx.wait();
            T_par[i] += chrono::high_resolution_clock::now() - T2;
            finished_fullerenes += output0_queue.get_size() + output1_queue.get_size();
            output0_queue.clear(device0_ctx);
            output1_queue.clear(device1_ctx);
            optimize_more = opt0_queue.get_size() >= opt0.isomer_capacity;
        }
        auto T3 = chrono::high_resolution_clock::now();
        generate_handle.wait();
        more_to_generate = generate_handle.get();
        opt0_queue.insert(input0, device0_ctx, ASYNC);
        opt1_queue.insert(input1, device1_ctx, ASYNC);
        device0_ctx.wait(); device1_ctx.wait();
        finished_fullerenes += output0_queue.get_size() + output1_queue.get_size();
        auto T4 = chrono::high_resolution_clock::now();
        if(more_to_generate) T_par[i] += T4 - T3;
        if(!more_to_generate){
            while(opt0_queue.get_size() > 0){
                isomerspace_forcefield::optimize¡BUSTER>(opt0, step, N*5, device0_ctx,
                    ↪ ASYNC);
                isomerspace_forcefield::optimize¡BUSTER>(opt1, step, N*5, device1_ctx,
                    ↪ ASYNC);
                output0_queue.push(opt0, device0_ctx, ASYNC);
                output1_queue.push(opt1, device1_ctx, ASYNC);
                opt0_queue.refill_batch(opt0, device0_ctx, ASYNC);
                opt1_queue.refill_batch(opt1, device1_ctx, ASYNC);
                device0_ctx.wait(); device1_ctx.wait();
            }
            for(int i = 0; i ¡ N*5; i += step){
                isomerspace_forcefield::optimize¡BUSTER>(opt0, step, N*5, device0_ctx,
                    ↪ ASYNC);
                isomerspace_forcefield::optimize¡BUSTER>(opt1, step, N*5, device1_ctx,
                    ↪ ASYNC);
            }
            output0_queue.push(opt0, device0_ctx, ASYNC);
            output1_queue.push(opt1, device1_ctx, ASYNC);
            device0_ctx.wait(); device1_ctx.wait();
            more_to_do = false;
        }
        T_ends[i] += chrono::high_resolution_clock::now() - T4;

    }
    }
    if(n_fullerenes > n_samples*10){
        auto total = (float)(mean(T_io)/1ns + mean(T_par)/1ns);
        std::cout ¡¡ std::fixed ¡¡ std::setprecision(2) ¡¡ N ¡¡ ",␣"¡¡ finished_fullerenes
            ↪ ¡¡ ",␣" ¡¡ (mean(T_par)/1ns)/total*100. ¡¡ "%,␣" ¡¡ (mean(T_io)/1ns)/total
            ↪ *100. ¡¡ "%,␣" ¡¡ (float)(mean(T_io)/1us+mean(T_par)/1us)/
            ↪ finished_fullerenes ¡¡ "us/isomer\n";
        out_file ¡¡ N ¡¡ ",␣"¡¡ n_fullerenes ¡¡ ",␣"¡¡ finished_fullerenes ¡¡ ",␣" ¡¡ mean
            ↪ (T_ends) /1ns ¡¡ ",␣" ¡¡ mean(T_par)/1ns ¡¡ ",␣" ¡¡ mean(T_io)/1ns ¡¡ "\n";
        out_file_std ¡¡ N ¡¡ ",␣"¡¡ n_fullerenes ¡¡ ",␣"¡¡ finished_fullerenes ¡¡ ",␣" ¡¡
            ↪ sdev(T_ends) /1ns ¡¡ ",␣" ¡¡ sdev(T_par)/1ns ¡¡ ",␣" ¡¡ sdev(T_io)/1ns ¡¡ "
            ↪ \n";}
    else{
        auto total = (float)(mean(T_io)/1ns + mean(T_par)/1ns + mean(T_ends)/1ns);
        std::cout ¡¡ std::fixed ¡¡ std::setprecision(2) ¡¡ N ¡¡ ",␣"¡¡ n_fullerenes ¡¡ ",␣
```

```
    ↪ " ¡¡ (mean(T_par)/1ns)/total*100. ¡¡ "%,␣" ¡¡ (mean(T_io)/1ns)/total*100. ¡
    ↪ ¡ "%,␣" ¡¡ (float)(mean(T_io)/1us+mean(T_par)/1us + mean(T_ends)/1us)/
    ↪ n_fullerenes ¡¡ "us/isomer\n";
  out_file ¡¡ N ¡¡ ",␣"¡¡ n_fullerenes ¡¡ ",␣"¡¡ n_fullerenes ¡¡ ",␣" ¡¡ mean(T_ends
    ↪ ) /1ns ¡¡ ",␣" ¡¡ mean(T_par)/1ns + mean(T_ends)/1ns ¡¡ ",␣" ¡¡ mean(T_io)
    ↪ /1ns ¡¡ "\n";
  out_file_std ¡¡ N ¡¡ ",␣"¡¡ n_fullerenes ¡¡ ",␣"¡¡ n_fullerenes ¡¡ ",␣" ¡¡ sdev(
    ↪ T_ends) /1ns ¡¡ ",␣" ¡¡ sdev(T_par)/1ns + sdev(T_ends)/1ns ¡¡ ",␣" ¡¡ sdev(
    ↪ T_io)/1ns ¡¡ "\n";}

    std::cout ¡¡ (float)finished_fullerenes / (float)(mean(T_par)/1ms) ¡¡ std::endl;
  }
  LaunchCtx::clear_allocations();
}
```

Listing C.9: *Benchmark Parallel Components*

```
#include "fullerenes/gpu/isomer_queue.hh"
#include "fullerenes/gpu/cuda_io.hh"
#include "fullerenes/gpu/kernels.hh"
#include "fullerenes/gpu/benchmark_functions.hh"
#include "numeric"
#include "random"
#include "filesystem"
using namespace gpu_kernels;

int main(int argc, char** argv){

    size_t N_start              = argc > 1 ? strtol(argv[1],0,0) : (size_t)20;       //
        ↪ Argument 1: Start of range of N
    size_t N_limit              = argc > 2 ? strtol(argv[2],0,0) : N_start;          //
        ↪ Argument 2: End of range of N
    size_t N_runs               = argc > 3 ? strtol(argv[3],0,0) : 3;                //
        ↪ Argument 3: Number of times to run experiment
    size_t warmup               = argc > 4 ? strtol(argv[4],0,0) : 0;                //
        ↪ Argument 4: Seconds to warmup GPU

    ofstream out_file   ("ParBenchmark_" + to_string(N_limit) + ".txt");
    ofstream out_std    ("ParBenchmark_STD_" + to_string(N_limit) + ".txt");
    out_file << "Generate, Samples, Update, Dual, Tutte, X0, Optimize \n";

    cuda_benchmark::warmup_kernel(warmup*1s);
    LaunchCtx ctx(0);
    for (size_t N = N_start; N < N_limit+1; N+=2)
    {
        if(N == 22) continue;
        Graph G;

        auto batch_size = isomerspace_forcefield::optimal_batch_size(N);
        auto n_fullerenes = (int)num_fullerenes.find(N)->second;
        auto sample_size = min(batch_size*1,n_fullerenes);
        if (n_fullerenes < batch_size){
            sample_size = n_fullerenes;
        } else if (n_fullerenes < batch_size*2){
            sample_size = batch_size;
        }else if (n_fullerenes < batch_size*3){
            sample_size = batch_size*2;
        }else if (n_fullerenes < batch_size*4){
            sample_size = batch_size*3;
        }else{
            sample_size = batch_size*4;
        }

        if (N == 22) continue;
        std::cout << sample_size << endl;

        bool more_to_generate = true;

        std::vector<std::chrono::nanoseconds>
            T_gens(N_runs),
            T_duals(N_runs),
            T_tuttes(N_runs),
            T_X0s(N_runs),
            T_opts(N_runs),
            T_flat(N_runs),
            T_queue(N_runs),
            T_io(N_runs);


        auto Nf = N/2 + 2;
        G.neighbours = neighbours_t(Nf, std::vector<node_t>(6));
        G.N = Nf;
```

```
auto path = "isomerspace_samples/dual_layout_" + to_string(N) + "_seed_42";
ifstream isomer_sample(path,std::ios::binary);
auto fsize = std::filesystem::file_size(path);
std::vector<device_node_t> input_buffer(fsize/sizeof(device_node_t));
auto available_samples = fsize / (Nf*6*sizeof(device_node_t));
isomer_sample.read(reinterpret_cast<char*>(input_buffer.data()), Nf*6*sizeof(
    ↪ device_node_t)*available_samples);

std::vector<int> random_IDs(available_samples);
std::iota(random_IDs.begin(), random_IDs.end(), 0);
std::shuffle(random_IDs.begin(), random_IDs.end(), std::mt19937{42});
std::vector<int> id_subset(random_IDs.begin(), random_IDs.begin()+sample_size);


auto finished_isomers = 0;
for (size_t l = 0; l < N_runs; l++)
{
    finished_isomers = 0;
    IsomerBatch batch0(N,sample_size,DEVICE_BUFFER,0);
    IsomerBatch batch1(N,sample_size,DEVICE_BUFFER,0);
    IsomerBatch batch2(N,sample_size,DEVICE_BUFFER,0);
    IsomerBatch h_batch(N,sample_size,HOST_BUFFER);
    LaunchCtx ctx(0);
    cuda_io::IsomerQueue isomer_q(N,0);
    cuda_io::IsomerQueue isomer_q_cubic(N,0);
    cuda_io::IsomerQueue OutQueue(N,0);
    OutQueue.resize(sample_size);
    isomer_q_cubic.resize(min(n_fullerenes,10000 + sample_size));
    isomer_q.resize(min(n_fullerenes,sample_size));
    for (int i = 0; i < sample_size; i++){
        for (size_t j = 0; j < Nf; j++){
            G.neighbours[j].clear();
            for (size_t k = 0; k < 6; k++) {
                auto u = input_buffer[id_subset[i]*Nf*6 + j*6 +k];
                if(u != UINT16_MAX) G.neighbours[j].push_back(u);
            }
        }
        isomer_q.insert(G, i);
    }
    batch0.clear();
    isomer_q.refill_batch(batch0);
    auto TDual = isomerspace_dual::time_spent();
    isomerspace_dual::dualize(batch0);
    auto TTutte = isomerspace_tutte::time_spent(); T_duals[l] += isomerspace_dual::
        ↪ time_spent() - TDual;
    isomerspace_tutte::tutte_layout(batch0);
    auto TX0 = isomerspace_X0::time_spent(); T_tuttes[l] += isomerspace_tutte::
        ↪ time_spent() - TTutte;
    isomerspace_X0::zero_order_geometry(batch0, 4.0);
    T_X0s[l] += isomerspace_X0::time_spent() - TX0;

    cuda_io::copy(batch1, batch0);
    while(isomer_q_cubic.get_size() < min(n_fullerenes,10000)){
        isomer_q_cubic.insert(batch1);
        cuda_io::copy(batch1, batch0);
    }
    auto TFF = high_resolution_clock::now();
    isomerspace_forcefield::optimize<BUSTER>(batch0,N*5,N*5);
    auto TFlat = high_resolution_clock::now(); T_opts[l] += TFlat - TFF;
    isomerspace_forcefield::optimize<FLATNESS_ENABLED>(batch1,N*5,N*5);
    T_flat[l] += high_resolution_clock::now() - TFlat;
    OutQueue.push(batch2,ctx, LaunchPolicy::SYNC);
    auto T0 = high_resolution_clock::now();
    auto j = isomer_q_cubic.get_size();
    if(n_fullerenes >= 10000){
        while(j > sample_size){
```

```
                auto T1 = high_resolution_clock::now();
                isomer_q_cubic.refill_batch(batch2,ctx, LaunchPolicy::SYNC);
                auto T2 = high_resolution_clock::now(); T_io[l] += T2 − T1;
                isomerspace_forcefield::optimize¡BUSTER>(batch2,N*0.5,N*5,ctx,
                    ↪ LaunchPolicy::SYNC);
                auto T3 = high_resolution_clock::now();
                OutQueue.push(batch2,ctx, LaunchPolicy::SYNC);
                finished_isomers += OutQueue.get_size();
                j = isomer_q_cubic.get_size();
                OutQueue.clear(ctx, LaunchPolicy::SYNC);
                T_io[l] += high_resolution_clock::now() − T3;
            }
        }else{
            while(finished_isomers ¡ n_fullerenes){
                auto T1 = high_resolution_clock::now();
                isomer_q_cubic.refill_batch(batch2);
                auto T2 = high_resolution_clock::now(); T_io[l] += T2 − T1;
                isomerspace_forcefield::optimize¡BUSTER>(batch2,N*0.5,N*5);
                auto T3 = high_resolution_clock::now();
                OutQueue.push(batch2);
                j = OutQueue.get_size();
                finished_isomers += j;
                OutQueue.clear();
                T_io[l] += high_resolution_clock::now() − T3;
            }
        }
        T_queue[l] += high_resolution_clock::now() − T0;
        ctx.wait();
        if(OutQueue.get_capacity()> 10000 + sample_size) std::cout ¡¡ ”Warning:␣OutQueue␣
            ↪ initial␣capacity␣exceeded” ¡¡ std::endl;
        //std::cout ¡¡ finished_isomers ¡¡ ”: ” ¡¡((high_resolution_clock::now() − Tio)/1
            ↪ us) ¡¡ ”us  ” ¡¡ (T_queue[l]/1us) ¡¡ ”us  ” ¡¡ (T_io[l]/1us) ¡¡ ”us  ” ¡¡ (
            ↪ Trefill/1us) ¡¡ ”us  ” ¡¡ (Tpush/1us) ¡¡ ”us  ” ¡¡ (Tget/1us) ¡¡ ”us  ” ¡¡
            ↪ (Tclear/1us)/(float)finished_isomers ¡¡ ”us  ” ¡¡ std::endl;

    }
    using namespace cuda_io;
    //Print out runtimes in us per isomer:
    std::cout ¡¡ N ¡¡ ”␣␣Dual:␣” ¡¡ std::fixed ¡¡ std::setprecision(2) ¡¡ (float)(mean(
        ↪ T_duals)/1us)/sample_size ¡¡ ”us␣␣Tutte:␣” ¡¡ std::fixed ¡¡ std::setprecision
        ↪ (2) ¡¡ (float)(mean(T_tuttes)/1us)/sample_size ¡¡ ”us␣␣X0:␣” ¡¡ std::fixed ¡¡
        ↪ std::setprecision(2) ¡¡ (float)(mean(T_X0s)/1us)/sample_size ¡¡ ”us␣␣Opt:␣” ¡¡
        ↪ std::fixed ¡¡ std::setprecision(2) ¡¡ (float)(mean(T_opts)/1us)/sample_size ¡¡
        ↪ ”us␣␣Flat:␣” ¡¡ std::fixed ¡¡ std::setprecision(2) ¡¡ (float)(mean(T_flat)/1us)
        ↪ /sample_size ¡¡ ”us␣␣Queue:␣” ¡¡ std::fixed ¡¡ std::setprecision(2) ¡¡ (float)(
        ↪ mean(T_queue)/1us)/finished_isomers ¡¡ ”us” ¡¡ std::fixed ¡¡ std::setprecision
        ↪ (2) ¡¡ ”␣␣IO:␣” ¡¡ (float)(mean(T_io)/1us)/finished_isomers ¡¡ ”us” ¡¡ std::
        ↪ endl;

    //Print out what fraction of the runtime that each component took:
    out_file ¡¡ N ¡¡ ”,␣”¡¡ sample_size ¡¡ ”,␣”¡¡ finished_isomers ¡¡ ”,␣” ¡¡ mean(T_gens)
        ↪ /1ns ¡¡ ”,␣” ¡¡ mean(T_duals)/1ns ¡¡”,␣” ¡¡  mean(T_X0s)/1ns ¡¡”,␣” ¡¡ mean(
        ↪ T_tuttes)/1ns¡¡ ”,␣” ¡¡ mean(T_opts)/1ns ¡¡  ”,␣” ¡¡ mean(T_flat)/1ns ¡¡  ”,␣”
        ↪ ¡¡ mean(T_queue)/1ns ¡¡ ”,␣” ¡¡ mean(T_io)/1ns ¡¡ ”\n”;
    out_std ¡¡ N ¡¡ ”,␣”¡¡ sample_size ¡¡ ”,␣” ¡¡ finished_isomers ¡¡ ”,␣” ¡¡ sdev(T_gens)
        ↪ /1ns ¡¡ ”,␣” ¡¡ sdev(T_duals)/1ns ¡¡”,␣” ¡¡  sdev(T_X0s)/1ns ¡¡”,␣” ¡¡ sdev(
        ↪ T_tuttes)/1ns¡¡ ”,␣” ¡¡ sdev(T_opts)/1ns ¡¡ ”,␣” ¡¡ sdev(T_flat)/1ns ¡¡ ”,␣” ¡¡
        ↪  sdev(T_queue)/1ns ¡¡ ”,␣” ¡¡ sdev(T_io)/1ns ¡¡ ”\n”;
  }

}
```

Listing C.10: *Benchmark Sequential Components*

```
#include "filesystem"
#include "random"
#include "numeric"
const std::unordered_map¡size_t,size_t> num_fullerenes =
    ↪ {{20,1},{22,0},{24,1},{26,1},{28,2},{30,3},{32,6},{34,6},{36,15},{38,17},{40,40},{42,45},{44,89},{46,116}
    ↪ ;
using namespace chrono;
using namespace chrono_literals;

int main(int argc, char** argv){
    const size_t N_limit                    = strtol(argv[1],0,0);      // Argument 1: Number of
        ↪ vertices N
    auto N_runs = 5;
    ofstream out_file("SeqBenchmark_" + to_string(N_limit) + ".txt");
    ofstream out_std("SeqBenchmark_STD_" + to_string(N_limit) + ".txt");
    out_file ¡¡ "Generate,␣Samples,␣Update,␣Dual,␣Tutte,␣X0,␣Optimize␣\n";
    for (size_t N = 20; N ¡ N_limit+1; N+=2)
    {
        if(N == 22) continue;
        BuckyGen::buckygen_queue Q = BuckyGen::start(N,false,false);

        auto sample_size = min(gpu_kernels::isomerspace_forcefield::optimal_batch_size(N,0),(
            ↪ int)num_fullerenes.find(N)->second);

        FullereneDual G;

        bool more_to_generate = true;

        std::vector¡std::chrono::nanoseconds>
            T_gens(N_runs,chrono::nanoseconds(0)),
            T_duals(N_runs,chrono::nanoseconds(0)),
            T_tuttes(N_runs,chrono::nanoseconds(0)),
            T_X0s(N_runs,chrono::nanoseconds(0)),
            T_opts(N_runs,chrono::nanoseconds(0)),
            T_polys(N_runs,chrono::nanoseconds(0));

        auto Nf = N /2 + 2;
        G.neighbours = neighbours_t(Nf, std::vector¡node_t>(6));
        G.N = Nf;

        auto path = "isomerspace_samples/dual_layout_" + to_string(N) + "_seed_42";
        ifstream isomer_sample(path,std::ios::binary);
        auto fsize = std::filesystem::file_size(path);
        std::vector¡device_node_t> input_buffer(fsize/sizeof(device_node_t));
        auto available_samples = fsize / (Nf*6*sizeof(device_node_t));
        isomer_sample.read(reinterpret_cast¡char*>(input_buffer.data()), Nf*6*
            ↪ available_samples*sizeof(device_node_t));

        std::vector¡int> random_IDs(available_samples);
        std::iota(random_IDs.begin(), random_IDs.end(), 0);
        std::shuffle(random_IDs.begin(), random_IDs.end(), std::mt19937{42});
        std::vector¡int> id_subset(random_IDs.begin(), random_IDs.begin()+sample_size);

        for (int l = 0; l ¡ N_runs; l++){
            for (int i = 0; i ¡ sample_size; ++i){
                for (size_t j = 0; j ¡ Nf; j++){
                    G.neighbours[j].clear();
                    for (size_t k = 0; k ¡ 6; k++) {
                        auto u = input_buffer[id_subset[i]*Nf*6 + j*6 +k];
                        if(u != UINT16_MAX) G.neighbours[j].push_back(u);
                    }
                }
                    auto T1 = high_resolution_clock::now();
                G.update();
                PlanarGraph pG = G.dual_graph();
                    auto T2 = high_resolution_clock::now(); T_duals[l] += T2 - T1;
```

```
            Polyhedron P(pG);
                auto T3 = high_resolution_clock::now(); T_polys[l] += T3 - T2;


            P.layout2d  = P.tutte_layout();
                auto T4 = high_resolution_clock::now(); T_tuttes[l] += T4 - T3;
            P.points    = P.zero_order_geometry();
                auto T5 = high_resolution_clock::now(); T_X0s[l] += T5 - T4;
            P.optimize();
                auto T6 = high_resolution_clock::now(); T_opts[l] += T6 - T5;
        }
    }
    using namespace cuda_io;
    out_file ¡¡ N ¡¡ ",␣"¡¡ sample_size ¡¡ ",␣" ¡¡ mean(T_gens)/1ns ¡¡ ",␣"  ¡¡ mean(
        ↪ T_duals)/1ns ¡¡",␣" ¡¡  mean(T_X0s)/1ns ¡¡",␣" ¡¡ mean(T_tuttes)/1ns¡¡ ",␣" ¡¡
        ↪ mean(T_opts)/1ns ¡¡ ",␣"¡¡ mean(T_polys)/1ns ¡¡ "\n";
    out_std ¡¡ N ¡¡ ",␣"¡¡ sample_size ¡¡ ",␣" ¡¡ sdev(T_gens)/1ns ¡¡ ",␣"  ¡¡ sdev(
        ↪ T_duals)/1ns ¡¡",␣" ¡¡  sdev(T_X0s)/1ns ¡¡",␣" ¡¡ sdev(T_tuttes)/1ns¡¡ ",␣" ¡¡
        ↪ sdev(T_opts)/1ns ¡¡  ",␣" ¡¡ sdev(T_polys)/1ns¡¡"\n";
  }

}
```