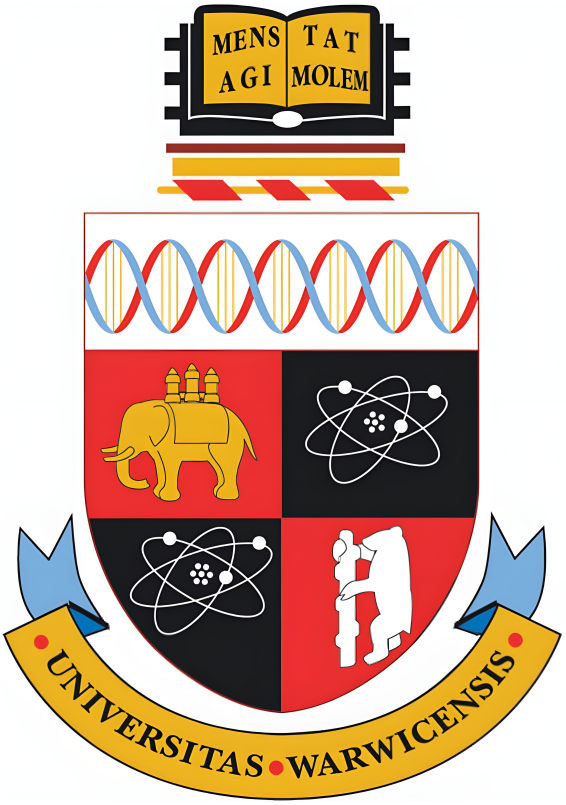


University of Warwick
Department of Computer Science
3rd Year Project

Optimising Machine Learning Algorithms for HPC Systems

Gerald Ramos

2145461

Supervised by Dr. Richard Kirk

Year of Study: 3rd Year MEng

Abstract

Vast quantities of machine learning algorithms have been often employed to solve problems commonly found in software applications used to generate prediction models and perform automatic classification of data. However, in the stride to improve these models in order to increase classification accuracy, larger datasets have to be used to increase the size of the training sets. This increase in training set size can increase the times taken to train models such as neural networks considerably. On the other hand, high performance computing systems such as those present on the Graph500 [14] benchmark tables combined with high level optimisation frameworks allow us to exploit parallelism which can be used in conjunction accelerate machine learning applications. In this work we present two optimised machine learning algorithms implemented in the high level optimisation framework, Kokkos [19]. The machine learning algorithms, K-Nearest Neighbours and Graph Convolutional Networks, were optimised such that both algorithms could run parallelised on CPUs and GPUs without having to write separate CPU-optimised and GPU-optimised versions of the code. Rather, we utilise Kokkos polymorphic memory and execution spaces [47] to optimise for both sets of hardware. Instrumentation and profiling were used to measure application performance on various train set sizes on multiple hardware configurations. Performance scaling was tested on up to 32 CPU cores and a single Nvidia A10 (24GB) GPU.

Keywords— High Performance Computing, Machine Learning, Performance Portability, Neural Networks, K-Nearest Neighbours, CPU, GPU

Acknowledgements

I would like to express my gratitude to my dissertation supervisor Dr. Richard Kirk for his continued support and works within the HPC area that contributed greatly throughout this project. I would also like to take this opportunity to thank Dr. Carter Edwards and his team at Sandia National Laboratories for their continued development and support of the Kokkos Core libraries alongside their lecture series which have helped significantly during the development of this application and provided strong insight into improving application performance by utilising all of the features that Kokkos has to offer.

The idea to optimise graph convolutional networks mainly came as inspiration from CS331 Neural Computing, therefore, many thanks to Dr. Weiren Yu for providing a clear understanding for topics such as neural network training. Finally, I would like to show my appreciation for my parents, for their continued support throughout my degree and this project.

Contents

List of Figures

Listings

List of Tables

1 Introduction

Machine learning algorithms are often utilised in a plethora of fields such as common statistical analysis found in scientific and financial fields. They may be also used to perform automation of tasks for businesses such as automated classification of products in the case of Amazon [82]. Specifically, we optimise K-Nearest Neighbours which is often used for classification tasks and market prediction [31][49]. Secondly, we look to optimise the training sequence of Graph Convolutional Neural Networks which can be utilised to solve the problems of training neural networks but in the context of graph structured data such as a citation-graph or a social network [34].

In the case of K-Nearest Neighbours, high dimensionality of data combined with vast datasets requires calculation of the euclidean distances between potentially billions of nodes in arbitrarily many dimensions. Furthermore, combined with the "Curse of Dimensionality" [87], with high dimensional data also comes the requirement for vast data set size to saturate the feature space which can increase memory and computational requirements significantly.

The necessity for performant hardware and efficient software to take full advantage of this hardware cannot be understated as the use of multiple CPU cores to parallelise processes whilst also providing performance portable optimisations allows for efficient implementation of these algorithms across a potentially vast array of hardware diverse high performance computing systems. This concept of many-core parallelism further extends into GPUs which are inherently designed to be optimised for parallel execution as a result of their device architecture which provides thousands upon thousands of simpler execution units to perform calculations such as subtraction and multiplication found in the euclidean distance calculations of K-Nearest Neighbours, in parallel in the case of device architectures such as Nvidia's CUDA [61]. This concept of single instruction multiple thread (SIMT) makes GPUs the optimal target hardware to optimise these algorithms for as many high performance computing systems employ multiple GPUs to perform these computations [86]. On the other hand, it has been found that the main bottleneck of utilising GPUs was not the speed of compute, however the saturation of memory bandwidth when performing memory copies from the host space (CPU memory) to the device space (GPU memory) [50]. Therefore, in our case a goal of our optimisations will be to reduce the overhead of memory bandwidth as much as possible by reducing the necessity to perform such memory copies in the first place by performing as much computation on the data already in device memory as possible before performing another memory copy.

The need for GPU-optimised Graph Convolutional Network training also comes as a result of graph data being represented in a non-linear fashion. That is, as graphs are inherently unstructured in terms of their layout, the vast majority of time taken when working with graph structured data comes as a result of non-linear memory access patterns [38]. GPUs are significantly more efficient at performing accesses to non-contiguous memory as a result of hardware features such as textured loads and the ability to group threading workloads in thread warps. Furthermore, dependent on the graph data structure format, the memory requirements for training on Graph Convolutional Networks can become infeasible. The original paper on Graph Convolutional Networks by Thomas Kipf utilised a matrix multiply between an adjacency matrix and a feature vector. Whilst adjacency matrices can be efficient in terms of access times they also have a space complexity of $O(|V|^2)$ which can make them unfeasible to store within GPU memory [34]. Hence, we utilise compressed row storage format [19] found in Kokkos as our graph representation to improve space complexity to $O(|E|)$ assuming our graph is sparse.

During prior investigation into already existing solutions to the problem aimed to be solved by this project it was found that many GPU-optimised Python based machine learning libraries such as PyTorch [72] and Tensorflow [1] already provide frameworks around GPU-optimised neural network training amongst other machine learning algorithms. On the other hand, these libraries only fully support Nvidia GPUs with minimal support for GPUs from other vendors. These libraries were also being limited by language dependent overhead of being written in an interpreted language as opposed to a compiled one. Hence, by implementing our algorithm in C++ utilising the performance portable framework, Kokkos, we remove this language dependent overhead whilst also laying the groundwork for our application to support accelerator hardware from other vendors such as AMD in the future [54]. Additionally, we also investigated the use of library optimised neural network training such as Nvidia CUDA Basic Linear Algebra Subprogram (BLAS) [56] which is a highly optimised library for performing common matrix operations and CUDA Deep Neural Networks (CuDNN) [58] which implement the activation functions for our Graph Convolutional Networks for us. The main rationale behind doing so is the fact that we can compare the performance of our performance portable Kokkos implemented kernels to a vendor specific library implemented function.

2 Background & Literature Review

Several of the tools utilised for this project and the rationale behind their requirement are explained within this section. Additionally, we further look into the specifics behind some of the mathematics used in order to understand the implementation of our algorithms from both an application design perspective alongside an optimisation orientated perspective.

2.1 Machine Learning Algorithms

A common operation utilised by both K-Nearest Neighbours (KNNs) and Graph Convolutional Networks (GCNs) are matrix-vector or matrix-matrix multiplies which occur very frequently in the area of machine learning and are quite prominent within the field of linear algebra. There already exists several highly optimised library functions such as LAPACK [6] and BLAS [9] which provide highly optimised implementations of these functions. However, these libraries are not always optimised for parallel execution on many-core systems such as GPUs. Therefore, we look to implement our own matrix-vector and matrix-matrix multiplies in order to take full advantage of the parallelism whilst also ensuring that our implementation is performance portable across multiple hardware configurations. In general the matrix-vector multiply can be defined as follows:

$$y = Ax \tag{2.1}$$

Which in matrix notation can be defined as:

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \tag{2.2}$$

Where A is an $m \times n$ matrix, x is an $n \times 1$ vector and y is an $m \times 1$ vector. The matrix-matrix multiply can be defined similarly:

$$C = AB \tag{2.3}$$

Which in matrix notation can be defined as:

$$\begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1p} \\ c_{21} & c_{22} & \cdots & c_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mp} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{bmatrix} \tag{2.4}$$

Where A is an $m \times n$ matrix, B is an $n \times p$ matrix and C is an $m \times p$ matrix. These operations are often used in the forward and backward propagation of neural networks and the calculation of euclidean distances in K-Nearest Neighbours. Additionally, the matrix-matrix multiply is also used in the calculation of the adjacency matrix and feature matrix multiplication in Graph Convolutional Networks to calculate the node aggregation. The matrix-vector and matrix-matrix multiplies can also be parallelised which can be performed on many-core systems such as GPUs. This can increase performance significantly as in terms of the matrix-matrix multiply, the time complexity of the operation is $O(n^3)$ which can be reduced to $O(n^3/p)$ where p is the number of threads available to perform the operation. However, a main bottleneck of performing matrix operations with graph

structured data is the non-linear memory access patterns which can be found in the adjacency matrix and feature matrix multiplication in Graph Convolutional Networks [34]. This is due to the fact that the adjacency matrix is often represented as a sparse matrix which can lead to non-contiguous memory accesses. The assumption being that in the vast majority of real-world graphs such as social networks, the number of edges is significantly less than the number of nodes [55]. This often leads to operations such as the sparse matrix-vector and sparse matrix-dense matrix multiplies being memory bound as opposed to compute bound [38], especially for large graphs. Matrix operations will therefore be one of the main areas of focus for optimisation in this project.

2.2 Optimisation Frameworks

Before we delve into the specifics of the optimisation framework, Kokkos, we must first understand the concept of performance portability. Performance portability is the ability to write code once and have it run efficiently on multiple hardware configurations without having to write separate code for each configuration. That is the concept of "write once, perform anywhere" but for high performance computing systems.

There are several several reasons as to why measuring performance portability in the space of high performance computing is vital in the development of optimised parallel applications. One of the main reasons is the fact that the performance of an application can vary significantly between different hardware configurations. Secondly, when taking into consideration the required time to develop an application, an issue that arises is often the need to write separate code for each hardware configuration. A measure of performance portability is the total lines of code required to write an application to be performant across multiple hardware configurations vs the total lines of code required to write an application to be performant on a single hardware configuration. This can be measured by the following formula:

$$\text{Performance Portability} = \frac{\text{LOC for Single Hardware Configuration}}{\text{LOC for Multiple Hardware Configurations}} \tag{2.5}$$

In other literature, multiple definitions exist for performance portability such as the ability to achieve a certain percentage of the peak performance of a hardware configuration across multiple hardware configurations [18]. This definition is used more often in the context of high performance computing as it takes into consideration the peak performance of an application as a harmonic mean across each configuration and applies a score of 0 if the application fails on any of the measured platforms [73][18]. This definition is more useful in our context and is defined as follows:

$$\mathbf{\Phi}(a, p, H) = \begin{cases} \frac{|H|}{\sum_{i \in H} \frac{1}{e_i(a,p)}} & \text{if all } h \in H \text{ pass} \\ 0 & \text{if any } h \in H \text{ fails} \end{cases} \tag{2.6}$$

where a is the application, p is the problem, H is the set of hardware configurations, $e_i(a, p)$ is the efficiency of the application on hardware configuration i and $|H|$ is the number of hardware configurations in H [73]. We utilise this measure as our main metric for performance portability when considering which optimisation framework to use for this project. Additionally, note that we may also have to take into account that certain hardware may have access to hardware specific features which may not be available on devices from different vendors or different generations of devices from the same vendor. This is especially prevalent in the case of matrix operations on newer Nvidia GPUs which have access to Tensor Cores which can significantly speed up matrix operations which will be discussed later [62].

Finally, one must also consider the fact that the field of high performance computing is moving towards heterogeneous systems which utilise multiple different types of hardware devices to perform computations [86]. One of the main pitfalls within the area of high performance computing is the fact when attempting to optimise an application for multiple hardware configurations, the application can often become less performant on a single specific hardware configuration. That is, the difference in performance between the observed and achievable performance can be significant when utilising performance portable frameworks. Several papers have observed this discrepancy across multiple applications and a diverse array of frameworks compared to vendor specific libraries such as in the case of optimisation of a Heat-Conduction Mini Application [18]. This discrepancy was observed to differ greatly from one optimisation framework to another. As an example, the observed performance of the Heat-Conduction Mini Application when utilising the performance portable framework, OpenACC, was measured to have an abstraction overhead of between 10% and 20% when compared to the achievable performance of the application written directly in CUDA [18].

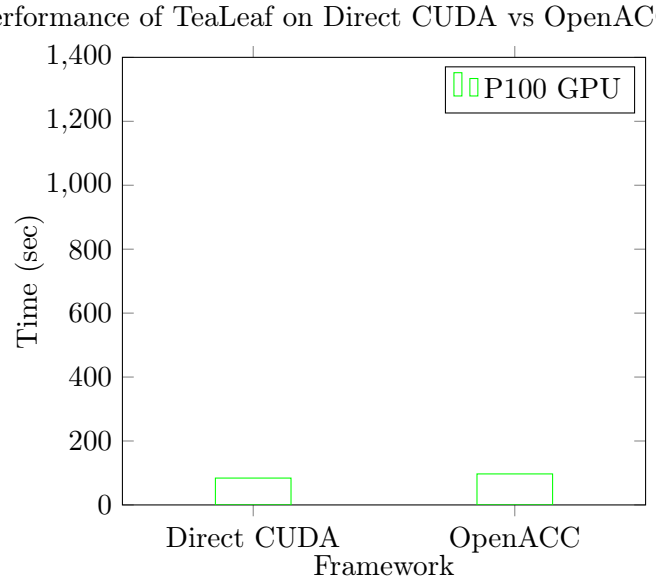


Figure 2.1. Performance of TeaLeaf on Direct CUDA vs OpenACC on 1000^2 [18]

Before selecting Kokkos as the optimisation framework for this project, several other performance portable C++ frameworks were considered such as RAJA [8] and SYCL [23]. However, Kokkos was selected as the optimisation framework for this project as it provides a vast array of features which are not present in other frameworks such as the ability to perform parallelisation across multiple hardware configurations through the use of polymorphic memory and execution spaces [47]. This allows for the same code to be run on CPUs and GPUs without having to write separate CPU-optimised and GPU-optimised versions of the code. Additionally, Kokkos provides a vast array of parallel algorithms which can be used to perform common parallel operations such as reductions and scans [46]. Kokkos also provides a vast array of parallel data structures which were vital for the efficient implementation of the Graph Convolutional Networks such as static compressed row storage format graphs [19] which was used to represent the graph data structure. Finally, Kokkos was also found to be highly performant when compared to other performance portable frameworks such as RAJA and SYCL as it was found to have a lower abstraction overhead when compared to other frameworks such as OpenACC for programming on GPUs, achieving up to 90% of the achievable performance compared to programming directly in CUDA in specific scenarios [18].

2.3 Kokkos

Our chosen performance portable optimisation framework, Kokkos, is a C++ library which provides programming abstractions for writing performance portable code across multiple hardware configurations. This library is currently being developed by Sandia National Laboratories and is being used in a vast array of high performance computing applications [39] such as problems in the areas of computational fluid dynamics and molecular dynamics simulations [41].

Figure 2.2. Kokkos Logo

As previously mentioned, we selected Kokkos for our optimisation framework mainly as a result of the necessity to have access to "write once, perform anywhere" code, abstraction of polymorphic data structures such as Kokkos Views and Static CRS Graphs and finally, the abstractions of parallel algorithms such as reductions and scans alongside its approach to thread level parallelism [52].

We make utilisation of the Kokkos lecture series [44] to understand the concepts of abstractions in regards to parallelism and memory spaces made by Kokkos in order to achieve maximal performance on both CPUs and GPUs available to us by utilising the Kokkos framework. Additionally, they also provide an array of examples and tutorials through the use of their documentation [39] which were vital in the development of our applications.
The largest contributor to the performance portability of Kokkos is the vast range of support available for targetted architectures within the high performance computing space. Kokkos provides support for a vast array of hardware configurations such as CPUs, GPUs and FPGAs [39]. Furthermore, the Kokkos Core libraries currently support CUDA, HIP, OpenMP, SYCL and HPX memory and execution spaces [47] which allows for simplified integration of Kokkos into existing HPC applications.

HIP or Heterogeneous-Compute Interface for Portability is a C++ runtime API and kernel language which allows for the porting of CUDA applications to run on AMD GPUs [5]. Kokkos currently provides an experimental HIP backend. An additional benefit of this is the fact that by implementing our applications in Kokkos, we lay the groundwork for our applications to be run on AMD GPUs in the future [54]. This is especially prevalent as whilst Nvidia GPUs are currently the most popular choice for high performance computing systems [86], AMD GPUs are also being used in a vast array of systems such as the Frontier Supercomputer at Oak Ridge National Laboratories [48].

2.3.1 OpenMP

We make utilisation of OpenMP running within the backend of Kokkos to parallelise our applications on CPUs. OpenMP is a widely used API for parallel programming in C, C++ and Fortran [10]. The library itself provides a multitude of directives which can be used to declare parallel regions, parallelelise loops and perform reductions amongst other operations. Of which, are all available to us through the Kokkos framework [39]. The main benefit of using OpenMP in conjunction with Kokkos is the fact that OpenMP is widely supported across multiple compilers and operating systems [10].

Writing parallel regions of code utilising Kokkos with OpenMP is as simple as adding a single directive to the code. For example, to parallelise a loop in C++ using OpenMP, one would write the following code:

```
#pragma omp parallel for
for (int i = 0; i < N; i++) {
    // perform some operation
}
```

Listing 2.1: Example of Parallel Loop in C++ using OpenMP

This code will parallelise the loop across all available CPU cores set by the OpenMP runtime. On the other hand, parallelising in Kokkos is almost identical.

```
Kokkos::parallel_for("ParallelLoop", N, KOKKOS_LAMBDA (const int i) {
    // perform some operation
});
```

Listing 2.2: Example of Parallel Loop in C++ using Kokkos

In fact, both loops will be parallelised utilising the same OpenMP runtime in the backend of Kokkos. In theory, the performance of both loops should be almost identical when taking into account the abstraction overhead of Kokkos. However, this may not always be the case as this is but a very simple example. In the case of more complex applications, the performance of the Kokkos loop may differ significantly from the OpenMP loop, especially when considering the performance of Kokkos on GPUs [18] as will be discussed later.

2.3.2 CUDA

CUDA or Compute Unified Device Architecture is a parallel computing platform for GPUs and application programming interface model created by Nvidia [61]. Similar to how Kokkos utilises OpenMP to parallelise applications on CPUs, Kokkos also provides a CUDA backend which allows for the parallelisation of applications on Nvidia GPUs [19]. Furthermore, Kokkos provides several abstractions for thread level parallelism on GPUs such as the "Kokkos::TeamPolicy" which allows for the parallelisation of applications across multiple thread blocks and warps [39]. This utilisation of hierarchical parallelism is vital for the efficient parallelisation of applications by ensuring that the maximum number of threads are utilised on the GPU combined with efficient memory layouts and memory access patterns [47].

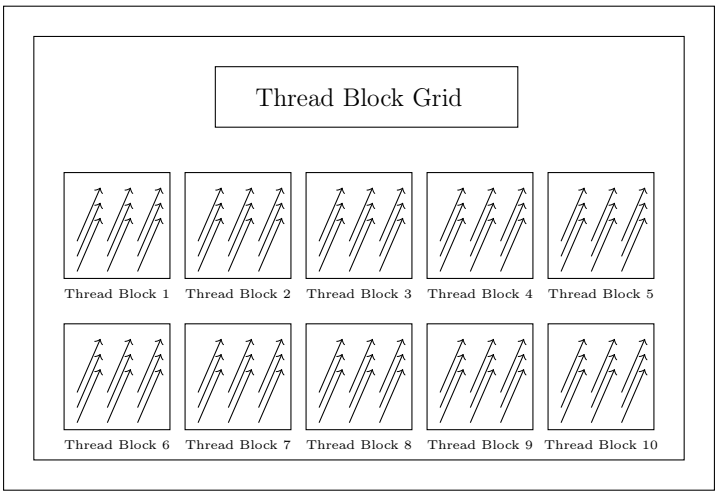


Figure 2.3. SIMT Execution Model on GPUs [61]

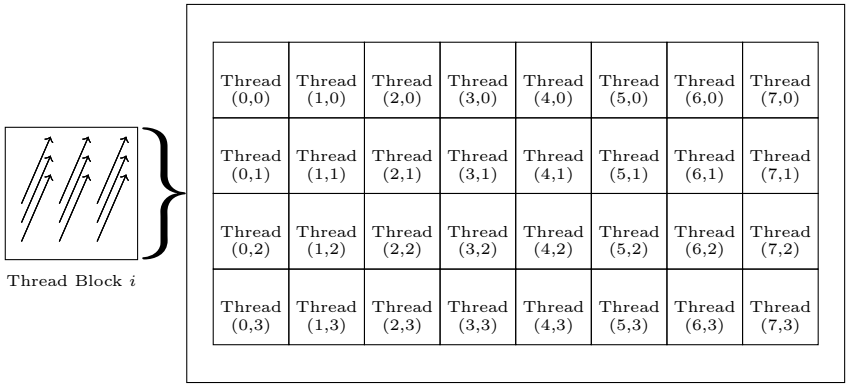


Figure 2.4. Thread Blocks Assigned 32-Thread Vector Lanes (warps) [61]

The execution model of GPUs is based on the concept of Single Instruction Multiple Thread (SIMT) [61]. From a conceptual standpoint, the GPU is divided into multiple thread blocks which are then further divided into warps as shown in Figure 2.4. Each thread block is assigned a number of warps which are then further divided into 32-thread vector lanes for Nvidia GPUs which are executed in parallel. Each thread block is then assigned to a Streaming Multiprocessor (SM) on the GPU which is responsible for the execution of the thread block. The SM then schedules the warps to be executed on the GPU cores.

Additionally, within each 32-thread vector lane (warp), each thread execution maps to a single core on the GPU. For Nvidia GPUs, these CUDA cores will then execute the workload in parallel across all threads in the warp. This execution model is vital for the efficient parallelisation of applications on GPUs as it allows for the maximum number of threads to be utilised whilst also ensuring that the memory access patterns are efficient and that the maximum number of cores are utilised on the GPU [61].

The main benefit of utilising Kokkos with CUDA is the many abstractions provided for implementing complex parallel algorithms that make full utilisation of the SIMT execution model on GPUs. This can be seen in many of the Kokkos execution policies, these policies control the execution of these parallel algorithms across multiple hardware configurations such as CPUs and GPUs [39].

Execution Policy	**CPU-Level**	**GPU-Level**
Kokkos::TeamPolicy	OpenMP Thread Teams	CUDA Blocks
Kokkos::TeamThreadRange	OpenMP Threads	CUDA Threads
Kokkos::TeamVectorRange	OpenMP SIMD	CUDA Warps
Kokkos::ThreadVectorRange	OpenMP SIMD	CUDA Warps

Table 2.1: Kokkos Hierarchical Execution Policies for CPUs and GPUs [39]

OpenMP Single Instruction Multiple Data (SIMD) is a directive which allows for the automatic vectorisation of loops on CPUs [10]. Effectively, this allows for vector operations via the use of SIMD registers on CPUs via Advanced Vector Extensions (AVX) or Streaming SIMD Extensions (SSE) [32] on x86-64 platforms. Previously, as AVX Intrinsics and SSE Intrinsics were specific only to certain x86-64 platforms, there was concern that the SIMD execution model and its subsequent optimisations would not be performance portable. However, with the advent of Kokkos, the SIMD execution model can be utilised across CPUs and GPUs. Essentially, at the "Kokkos::ThreadVectorRange" level, the SIMD execution model is made use of on CPUs via the utilisation of the OpenMP SIMD directive "#simd ivdep" [39] or "Ignore Vector Dependecies" which allows for compilers to make the assumption that there are no loop-carried dependencies within the loop and that the loop can be safely vectorised. On the other hand, when compiled for GPUs,

the "Kokkos::ThreadVectorRange" execution policy is mapped to CUDA Warps within 32-thread vector lanes [39].

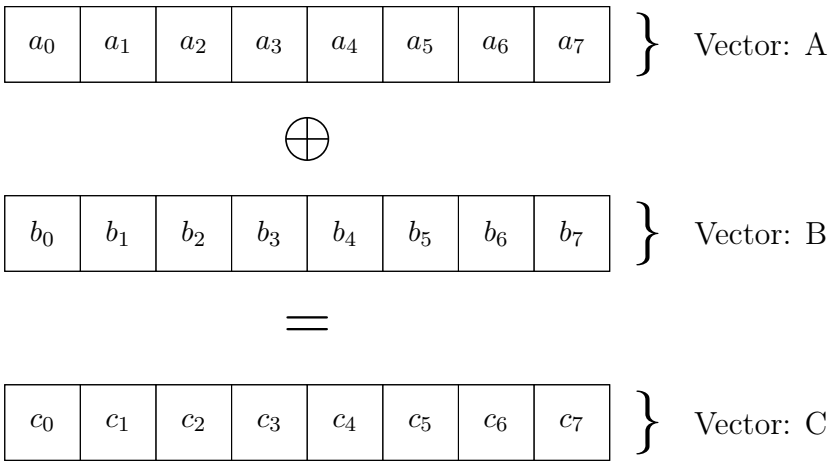


Figure 2.5. Vectorised Operation on Intel CPU AVX2 SIMD Execution Model [32]

Essentially, Kokkos allows for hardware specific optimisations to be made without loss in performance portability. Additionally, the Kokkos programming model enables powerful abstractions for hierarchical parallelism which can apply to a plethora of hardware configurations to exploit full utilisation of system hardware [39].

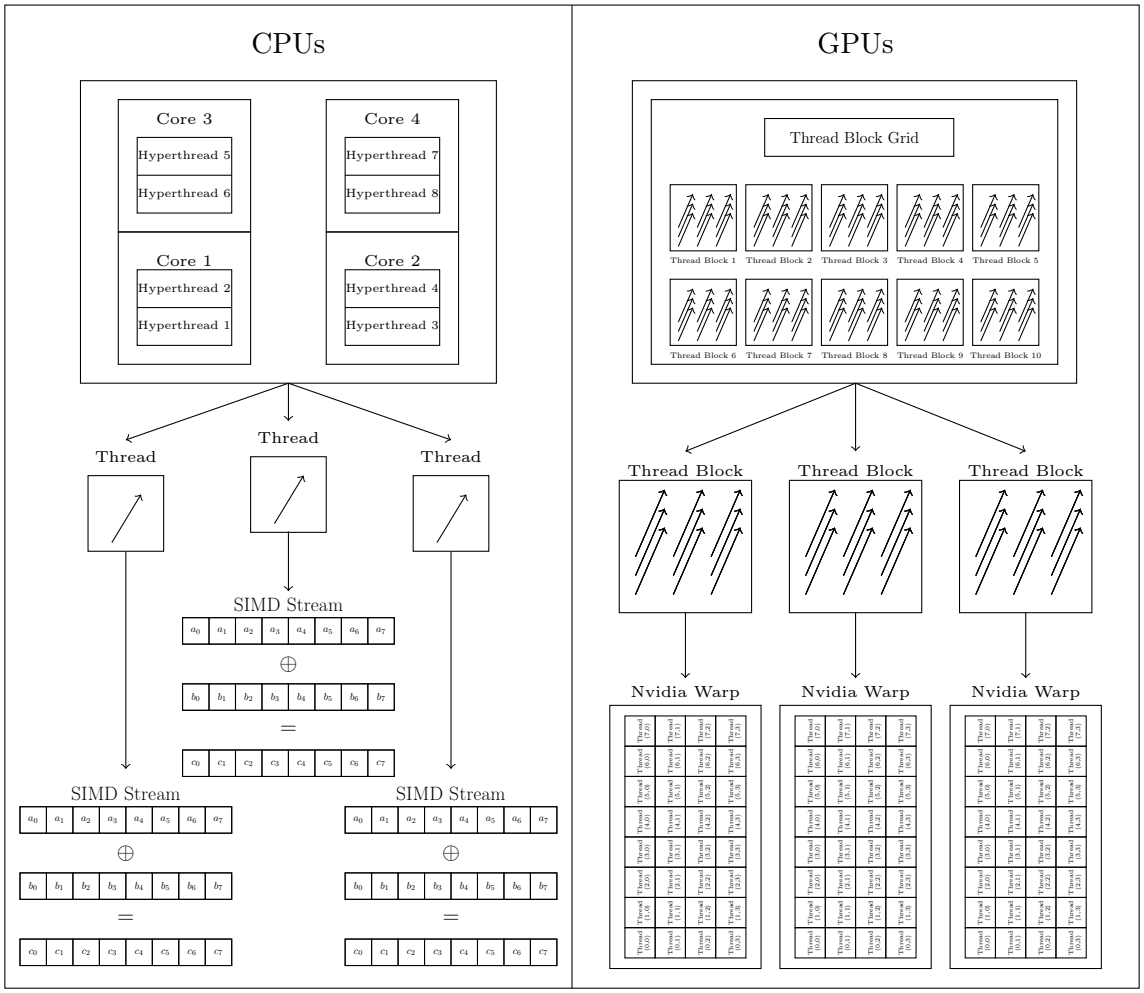


Figure 2.6. Example 3-Layer Hierarchical Parallelism in Kokkos [39]

3 K-Nearest Neighbours

K-Nearest Neighbours (KNN) is a non-parametric classification algorithm which can be used for tasks such as classification and regression [31]. The algorithm itself is relatively simple, given a dataset of points in a multi-dimensional space, the algorithm classifies a new point based on the majority class of its K nearest neighbours. The size of the model itself is dependent on the size of the dataset n as the algorithm does not require training data to be stored [3].

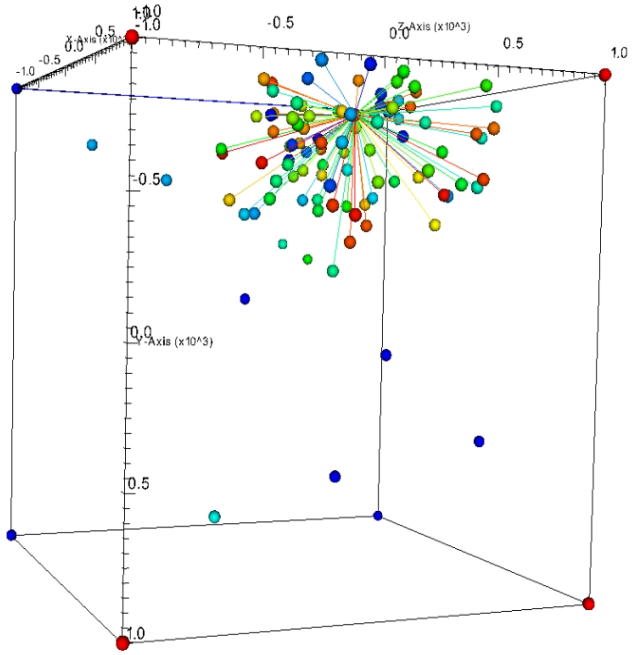


Figure 3.1. K-Nearest Neighbours Visualisation on a 3D Dataset

The main basis of the algorithm is built upon the concept of the Lipschitzness assumption which states that points which are close to each other in the input space are likely to have similar output values [3]. We selected KNN as the first algorithm to optimise via the use of performance portable frameworks. The rationale behind doing so centres around the following facts concerning the implementation of KNN in practise:

- The algorithm is highly parallelisable as the distance calculation between points can be done in parallel.
- The algorithm is highly memory bound as the distance calculation between points requires a large amount of memory access.
- The algorithm is highly data dependent as the performance of the algorithm is dependent on the size of the dataset and the number of dimensions.

The main goal of this chapter is to provide an overview of the implementation of KNN on CPUs and GPUs using the Kokkos framework. Keeping the following points in mind in regards to the algorithm. Its time complexity can be measured in our case as $O(ncd)$ where n is the number of points in the dataset, c is the number of feature vectors to classify and d is the number of dimensions in the dataset. This ensures that for large enough problem sizes, that the benefits of parallelism is more than likely going to be realised in increasing orders of dataset size and dimensions.

3.1 Design

We implement the KNN algorithm itself utilising the Euclidean distance, otherwise known as the L2 norm, as our measure of distance between points in the dataset, as it one of the most common distance metric used in KNN [31]. The Euclidean distance between two points p and q in a d-dimensional space is defined as follows:

$$\text{Euclidean Distance} = \sqrt{\sum_{i=1}^{d}(p_i - q_i)^2} \tag{3.1}$$

When implementing the KNN algorithm, we must also consider the fact that sorting the distances between the query point and all points in the dataset can be a computationally expensive operation. Aditionally, the sorting operation itself must utilise a stable sorting algorithm as the algorithm must be able to handle the case where two points have the same distance from the query point. In our implementation, we utilise the Merge Sort algorithm as it is a stable sorting algorithm with a time complexity of $O(n \log n)$ [15]. The algorithm itself is highly parallelisable as the sorting operation can be done in parallel across multiple threads on CPUs. The KNN algorithm itself can be summarised as follows:

Algorithm 1 K-Nearest Neighbours Algorithm

Input: Dataset X, Query Point q, Number of Neighbours K
Output: Class of Query Point q

Procedure:

1.) Calculate the Euclidean distance between the query point q and all points in the dataset X
2.) Merge Sort by key (distances) value (classifier) pairs in ascending order
3.) Select the first K points in the sorted distances
4.) Return the majority class of the K points

This implementation of the KNN algorithm is in essence quite brute force as it calculates the distance between the query point and all points in the dataset. However, this is a necessary step as the algorithm itself sets the value of K to be an arbitrary hyperparameter which can be tuned to achieve the best classification performance on the dataset. The value K could even be set to the size of the dataset itself, which necessitates the need for the algorithm to calculate every Euclidean distance.

Our main goal of this version of the KNN algorithm is to provide a baseline for the performance of the algorithm across different version and hardware configurations and core counts. We also utilise this version to demonstrate the benefits and scalability of parallelising applications whilst implementing these algorithms in performance portable frameworks such as Kokkos. To further demonstrate the benefits of parallelism, we intend to test the performance of our algorithm implementation across a vast array of problem sizes scaling the range of both dataset size and dimensions.

Dimensions Dataset Size	1M	10M	100M
10	40MB	400M	4GB
100	400MB	4GB	40GB
1000	4GB	40GB	400GB

Table 3.1: Problem Size and Memory Usage for KNN

3.2 Implementation

The implementation itself will take 5 key input parameters before running a simulation of classifying arbitrarily many points within the dataset. The output of the simulation will be the time taken to classify every point in the dataset. Alongside the profiling of the algorithm, we will also output containing each points feature vector and the class of the point. The 5 key input parameters are as follows:

- **Dataset Size:** The number of points in the dataset.
- **Dimensions:** The number of dimensions in the dataset.
- **Query Size:** The number of points to classify in the dataset.
- **Number of Neighbours:** The number of neighbours to classify the query points.
- **Number of Classes:** The number of classes in the dataset.

The N-Dimensional Space use to represent the dataset appeared within our implementation as a 2-Dimensional array of size $n \times d$. The feature vectors we we would classify would also be represented as a 2-Dimensional array of size $q \times d$ where q is the number of points to classify. We utilise 32-bit floating point values to represent the feature vectors of the points in the dataset and the query points to classify. The classes of the points in the dataset are represented as 32-bit integers. For each classifier we store the class of the point in an array of size n. Finally, to store each distance between each query point and each point in the dataset, we store the distances in an array of size $n \times q$.

3.2.1 Kokkos Views

The Kokkos framework provides a powerful abstraction for multi-dimensional arrays called Kokkos Views [39]. Previously in the original implementation of the KNN algorithm, we simply utilised 2-Dimensional arrays to represent the dataset and the query points. However, by utilising Kokkos Views, we can take advantage of the abstractions provided by Kokkos to represent the dataset and query points in a more convenient manner.

The main benefit of utilising Kokkos Views is the fact that the views themselves can be mapped to different memory spaces such as CPU memory and GPU memory. They can even specify row-major or column-major memory layouts which can be beneficial for cases where in certain memory layouts for operations such as matrix transposes and matrix multiplications can be more efficient [47]. The Kokkos Views themselves can even be templated to support different access patterns such as random access, strided access and even unmanaged access to the underlying memory [47].

```
using namespace Kokkos;
#ifdef KOKKOS_ENABLE_CUDA
    #define MemSpace CudaSpace
    #define MemLayout LayoutRight
#endif
#ifdef KOKKOS_ENABLE_OPENMP
    #define MemSpace OpenMP
    #define MemLayout LayoutRight
#endif
#ifndef MemSpace
    #define MemSpace HostSpace
    #define MemLayout LayoutLeft
#endif
```

Listing 3.1: Kokkos View Memory Space and Layout Definitions Using Preprocessor Directives

```
// Allocate memory space for the arrays
// float ** represents a 2D array of floats
View<float**, MemLayout, MemSpace> nDimensionalSpacePartition( "
    nDimensionalSpace", D, N );
View<float**, MemLayout, MemSpace> queryPoint( "queryPoint", C, D );
View<float**, MemLayout, MemSpace> distances( "distances", C, N );
View<int**, MemLayout, HostSpace> classifier( "classifier", C, N );
View<int*, MemLayout, HostSpace> finalClasses( "finalClasses", C );
```

Listing 3.2: Kokkos Views for KNN Implementation

The code snippet above demonstrates the use of Kokkos Views to represent the dataset, query points, distances, classifier and final classes. Additionally, we could also specify a memory access pattern for the Kokkos Views to utilise. A prominent access pattern is "Kokkos::RandomAccess" which is beneficial for cases where the order of data access cannot be predicted in advance, and corresponds to a strided memory access pattern [47].

As demonstrated in listing 3, we can specify the memory space for the Kokkos Views to either be on the Host or on external devices such as GPUs. Furthermore, Kokkos Views themselves are managed by the Kokkos runtime which ensures that the memory is correctly allocated and deallocated when the Kokkos View goes out of scope or if the number of View pointers referencing that memory space becomes zero [39].

Kokkos View Arguments	Equivalent Specification
Kokkos::LayoutLeft	Column-Major
Kokkos::LayoutRight	Row-Major
Kokkos::RandomAccess	Strided Access
Kokkos::CudaSpace	CUDA Memory
Kokkos::Experimental::HIPSpace	HIP Memory
Kokkos::OpenMP	Host Memory
Kokkos::HostSpace	Host Memory

Table 3.2: Kokkos View Terminology and Equivalent C++ Specification

3.2.2 Dataset

The dataset itself is generated randomly using the C++ standard library random number generator. In addition to this, when the dataset is generated, all classes, query points, data points and classifiers are generated randomly. We also ensure that the dataset is generated in a manner such that the tests and the profiling of the algorithm are consistent across different runs of the algorithm. As such, we seed the random number generator with a fixed seed value to ensure that our results are reproducible primarily for checksum and profiling purposes.

We also ensure that the dataset is random such that the classifiers and feature vectors are not correlated in any way. This is to ensure that the KNN algorithm itself is not biased towards any particular class or feature vector in the dataset. In most real world scenarios, KNN would normally be run after a preprocessing step such as the utilisation of Kernel Principal Component Analysis (KPCA) to reduce the dimensionality of the dataset [81]. Furthermore, most real world datasets would have classes and feature vectors appear in "clusters" which would make the KNN algorithm more effective. Finally, due to the curse of dimensionality, large datasets are preferred as the algorithm itself is more effective when the number of dimensions is less than the number of points in the dataset [3].

3.2.3 Euclidean Distance Calculation

As stated previously, the definition of Euclidean Distance from Equation (3.1) is used to calculate the distance between the query point and all points in the dataset. However, the calculation of the Euclidean distance itself can be extremely computationally expensive as it requires the calculation of the square root of the sum of the squares of the differences [3]. To further exacerbate the issue, for vast datasets with many dimensions, the cost of memory access can be very high as the algorithm must access each feature vector in the dataset. To mitigate this issue, we first parallelise the calculation of the Euclidean distance amongst multiple threads on CPUs before considering extending our optimisations to run on GPUs.

```
// Calculate the Euclidean distance between the query point and all points in the dataset
// Loop interchange was a good idea here, as it allows for improved memory locality
float diff;
for (int i = 0; i < D; i++) {
  for (int j = 0; j < N; j++) {
    diff = queryPoint(repeats, i) - nDimensionalSpace(i, j);
    distances(j) = diff * diff + distances(j);
  }
}
```

Listing 3.3: Calculation of Euclidean Distance Between Query Point and Dataset

As demonstrated before in listing 2, we can apply Kokkos parallelisation directives utilising the "Kokkos::parallel_for" directive can be used to parallelise the outer loop of the Euclidean distance calculation. Similarly to the utilisation of OpenMP #pragma statements, the "Kokkos::parallel_for" directive can be used to parallelise the outer loop of the Euclidean distance calculation across multiple threads on CPUs. There further exists a plethora of Kokkos parallelisation directives which can be used to specialise the parallelisation of the algorithm for different levels of the parallelism hierarchy [39]. Additionally, we can make use of specialisation of certain access patterns and workload scheduling directives.

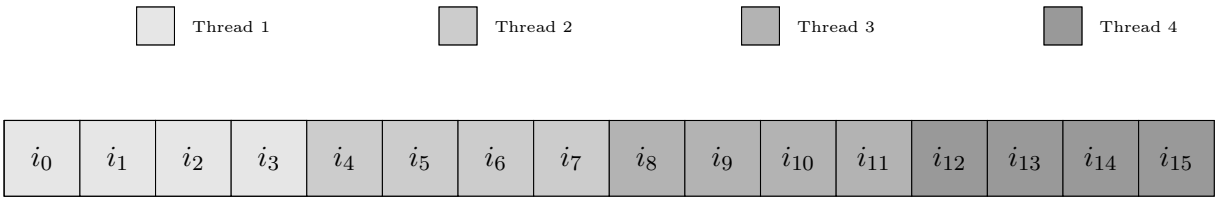


Figure 3.2. Workload Scheduling of Kokkos::Static Directive

The "Kokkos::Static" directive can be used to specify the workload scheduling of the parallelised loop by splitting each loop iteration evenly across the number of threads specified [39]. The benefits of utilising the "Kokkos::Static" directive is the fact that assuming that each iteration of the loop has the same workload, the workload can be evenly distributed across the threads in the parallelised loop. However, the "Kokkos::Static" directive is not suitable for cases where the workload of each iteration of the loop is not uniform. Additionally, if the range of the loop is particularly large, the "Kokkos::Static" directive may not be the most efficient directive to use. This is as a results of the fact that memory locality may be lost as the physical memory locations of the data may be far apart from each other in memory [84]. For this reason, the "Kokkos::Dynamic" directive may be more suitable for cases where the workload of each iteration of the loop is not uniform and the range of the loop is vast.

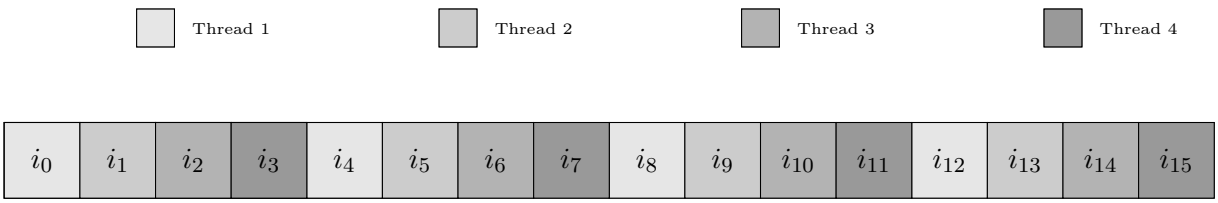


Figure 3.3. Workload Scheduling of Kokkos::Dynamic Directive (ideal)

The alternative Kokkos scheduling directive, "Kokkos::Dynamic", makes use of a work-stealing algorithm to dynamically assign work to threads as soon as they become available. This can give some guarantee that the workload of each thread is balanced assuming that each iteration workload may not be uniform. Also, as demonstrated in Figure 3.3, the locality of the memory access may be worse for each individual thread as they may have to access memory locations that are of length n apart from each other in memory, where n is the number of threads.

```
#define ScheduleType Schedule<Static>
using current_range_policy = RangePolicy<MemSpace, ScheduleType>;
parallel_for( "n_dimensional_euclid_calc",
  current_range_policy(0, numThreads), KOKKOS_LAMBDA ( int i ) {
  int problemSize = N/numThreads;
  int start = i * problemSize;
  int ending = (i + 1) * problemSize;
  if (i == numThreads - 1) {
      ending = N;
  }
  for (int k = 0; k < D; k++) {
    for (int j = start; j < ending; j++) {
      float diffSquare=queryPoint(repeats,k) - nDimensionalSpace(k,j);
      distances(j)+=diffSquare*diffSquare;
    }
  }
});
```

Listing 3.4: Kokkos Parallelisation of Euclidean Distance Calculation

Our first implementation of the optimised KNN algorithm divides the workload of the Euclidean distance calculation with a static scheduling directive across multiple threads, with the chunk size that each thread processes being equivalent to the number of points in the dataset divided by the number of threads. On the other hand, this implementation is not ideal for our use case for the following reasons:

- The memory locality of each thread may be poor as the memory locations of the data may be far apart from each other in memory.
- Not performance portable, as this implementation is only suitable for CPUs.
- Does not take advantage of the hierarchical parallelism provided by Kokkos.

To address these issues, we will further optimise the KNN algorithm by taking advantage of the hierarchical parallelism provided by Kokkos. As an aside, we may have to write a custom implementation of the Euclidean distance calculation to extend our optimisations to GPUs. Whilst Kokkos abstractions such as Kokkos Views and Kokkos parallelisation directives are powerful, they may not be sufficient to fully optimise algorithms for both CPUs and GPUs due to massive fundamental hardware architectural differences. As such, in some cases we may have to incorporate preprocessor directives to specialise the implementation of the algorithm for different hardware configurations.

3.2.4 Euclidean Distance Calculation on CPUs

Within our final implementation of the KNN algorithm, we optimise the Euclidean distance calculation further By utilising some abstractions of hierarchical parallelism provided by Kokkos. The "Kokkos::TeamPolicy" directive, allows developers to specify "Thread Teams", with each team consisting of a league of threads that can all be executed in parallel [39].

This is particularly useful, especially in cases where a large computational overhead comes as a result of initialisation of threads and the creation of thread teams [7]. Rather than creating a new thread team for each iteration of the loop, the "Kokkos::TeamPolicy" directive launches an initial team of threads which can be reused across multiple iterations of the loop.

```
// preprocessor directives to specify which memory space to use
#ifdef KOKKOS_ENABLE_OPENMP
// this forces a loop blocking protocol
long long numThreads = N/BLOCK_SIZE;
if (numThreads == 0) numThreads = 1;
using execution_policy = TeamPolicy<current_execution_space>;
using namespace Kokkos;
// utilise team policy for each point
parallel_for(execution_policy(C, AUTO),
  KOKKOS_LAMBDA (const execution_policy::member_type& team1) {
  long long repeats = team1.league_rank();

  // nested parallelism for each team
  parallel_for(TeamThreadRange(team1, numThreads),
    [=] (long long i) {
    int problemSize = N/numThreads;
    int start = i * problemSize;
    int ending = (i + 1) * problemSize;
    if (i == numThreads - 1) {
        ending = N;
    }
    for (int k = 0; k < D; k++) {
      parallel_for(ThreadVectorRange(team1, start, ending),
        [=] (int j) {
        float diffSquare = queryPoint(repeats,k) - nDimensionalSpace(k,j);
        distances(repeats,j) += diffSquare * diffSquare;
      });
    }
  });
});
#endif
```

Listing 3.5: Kokkos Parallelisation of Euclidean Distance Calculation on CPUs

Here, we make use of the "Kokkos::TeamPolicy" directive to launch thread teams for each point in the dataset. This is the finalised implementation of the loop for Euclidean distance calculation on CPUs. Important to note the fact of a loop blocking protocol being enforced [84], which ensures that memory accesses occur within memory locations that are within the range of a smaller predefined block of memory. This is particularly useful for cases where the memory access pattern is performed on a vast array of memory locations. For our case of the Euclidean distance calculation, the loop blocking protocol ensures that the memory access pattern occurs in blocks of 1024 using the #DEFINE preprocessor directive. On the other hand, we could also utilise the "lscpu" command to determine the optimal block size for the loop blocking protocol dependent on the size of the cache hierarchies [21].

3.2.5 Euclidean Distance Calculation on GPUs

Extending our optimisations to GPUs, we must consider the fact that the memory hierarchy of GPUs is vastly different from that of CPUs. As such, we must write a custom implementation of the Euclidean distance calculation to take advantage of the massive parallelism provided by GPUs. The "Kokkos::CudaSpace" directive can be used to specify that the Kokkos View is to be allocated in GPU memory and that a parallelised loop is to be executed on the GPU [39].

We also take advantage of the fact that GPUs have significantly faster memory access times when considering access to non-local memory locations. In recent development of GPUs, the memory access times to non-local memory locations have been significantly reduced due to the development of technologies such as High Bandwidth Memory (HBM) [90] and GDDR6 memory on the Nvidia A10 [62]. This is also present within the Kokkos framework, via the utilisation of the "Kokkos::RandomAccess" directive for Kokkos Views, which can be used to specify that the memory access pattern of the Kokkos View is strided [39].

```
#ifdef KOKKOS_ENABLE_CUDA
parallel_for( "n_dimensional_euclid_calc",
  current_range_policy(0, C * N), KOKKOS_LAMBDA (long long i) {
  float diff = 0.0f;
  long long repeats = i / N;
  long long j = i % N;
  for (int k = 0; k < endIndex; k++) {
    float diffSquare = queryPoint(repeats, k) - nDimensionalPartition(k, j);
    diff += diffSquare * diffSquare;
  }
  distances(repeats,j) += diff;
});
#endif
```

Listing 3.6: Kokkos Parallelisation of Euclidean Distance Calculation on GPUs

3.2.6 Workload Partitioning

One of the key problems that we encountered when optimising the KNN algorithm for GPUs was the fact that the dataset could not fit within the memory size of most GPUs as opposed to the memory available on the host device space. Specifically, when testing on the Nvidia A10, the largest problem size of 100 million datapoints with 200 dimensions occupied 80GB of memory for the N-Dimensional Space alone. As such, we had to partition the dataset into smaller, 24GB chunks to fit within the memory of the GPU. This was done such that the GPU could work on a smaller partition within its device space whilst the rest of the dataset was stored in the host memory space. Furthermore, we utilise the Kokkos abstraction of polymorphic views to allow for the dataset to be partitioned utilising subviews and what are effectively 3-Dimensional arrays to represent the dataset.

```
  Kokkos::View<float**, Kokkos::LayoutRight, Kokkos::CudaSpace>
  nDimensionalSpacePartition( "nDimensionalSpacePartition", partitionD, N );
  Kokkos::View<float***, Kokkos::LayoutRight, Kokkos::HostSpace
  nDimensionalSpace( "nDimensionalSpace", partitions, partitionD, N );
```

Listing 3.7: Kokkos Views for Workload Partitioning of Dataset

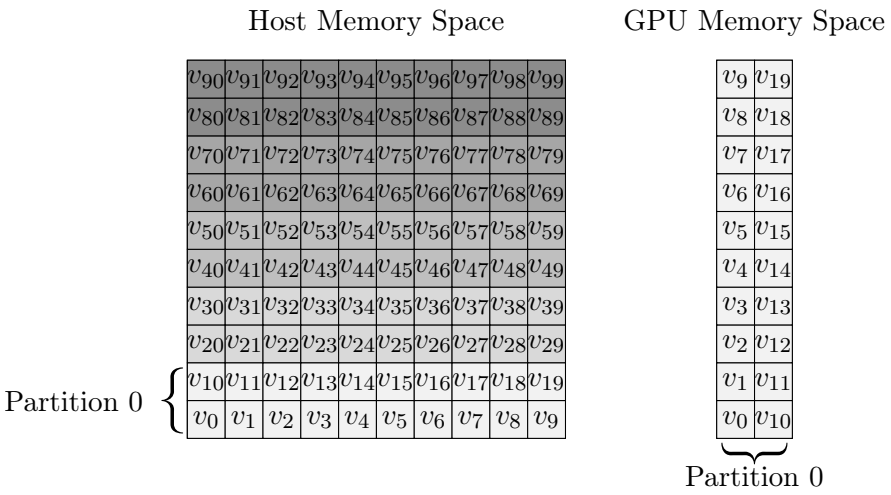


Figure 3.4. Workload Partitioning of Dataset for CPU and GPU Memory Spaces

The host memory denoted by "Kokkos::HostSpace" is used to store partitions for the entire dataset and where all of N-Dimensional Space is stored. "partitionD" denotes the maximal number of dimensions for each feature vector that is able to fit within the memory of the GPU with N many datapoints and hence the number of partitions can be calculated as $\frac{D}{partitionD}$. Performing a "deep_copy" operation from the host memory to the device memory, is made quite simply through the use of "Kokkos::Views" and "Kokkos::subviews" [47]. The following code snippet demonstrates the partitioning of the datset via the creation of a subviews with a partition for the GPU to work on which is then copied to the GPU memory space.

```
auto currWorkload = Kokkos::subview(nDimensionalSpace,
                                    workloadPartitioning,
                                    Kokkos::ALL(),
                                    Kokkos::ALL());
Kokkos::deep_copy(nDimensionalPartition, currWorkload);
```

Listing 3.8: Deep Copy Operation for Workload Partitioning of Dataset

In this scenario, we make use of the Kokkos::subview directive to create a subview of the dataset to convert the 3D "nDimensionalSpace" array into a 2D array "nDimensionalSpacePartition" which contains a 2D partition view that contains $partitionD * N$ space. The "workloadPartitioning" selects which partition to store whilst "Kokkos::ALL()" acts as a function macro to select all values across that dimension. Finally, the "Kokkos::deep_copy" operation is ubiquitous to Nvidia's CUDA "cudaMemcpy" function which is often used to copy data from the host memory to the device memory [60]. However, one must also acknowledge the implications of performing memory copies on the performance of the algorithm as it can be a significant bottleneck especially when the memory copies start to saturate the PCI-E bandwidth and host & device memory speeds [74].

3.2.7 Parallelised Sorting

The final optimisation that we make to the KNN algorithm is the parallelisation of the sorting of the classifiers by their associated distances. Originally, we utilised a single threaded merge sort algorithm to sort the classifiers. However, this was not ideal as eventually the sorting of the classifiers was a significant bottleneck in the algorithm. To address this issue, the parallelisation of sorting was performed by assigning each thread to sort a subset or partition of the classifier array before merging.

Algorithm 2 Parallelised Merge Sort

function ParallelMergeSort(classifiers, distances, numThreads)
  $partitionSize \leftarrow \frac{N}{numThreads}$
  $Start \leftarrow 0$
  $End \leftarrow partitionSize$
  for $i \leftarrow 0, numThreads$ **do**
    $Start \leftarrow i * partitionSize$
    $End \leftarrow (i+1) * partitionSize$
    $classifiersPartition \leftarrow$ SortByKey$(classifiers, distances[Start : End])$
  end for
  $classifiers \leftarrow$ ParallelMerge$(classifiers, distances)$ ▷ Merge the partitions
end function

Effectively, we divide and conquer amongst several threads to sort the classifiers by their associated distances. Merging the sorted subarrays can also be performed in parallel, as for every pair of subarrays, we can assign a thread to merge them until the entire array is sorted. In terms of algorithmic time complexity, we reduce the standard $O(n\log(n))$ time complexity of merge sort to $O(\frac{n}{p}\log(\frac{n}{p}))$ where p is the number of threads used to sort the array [11].

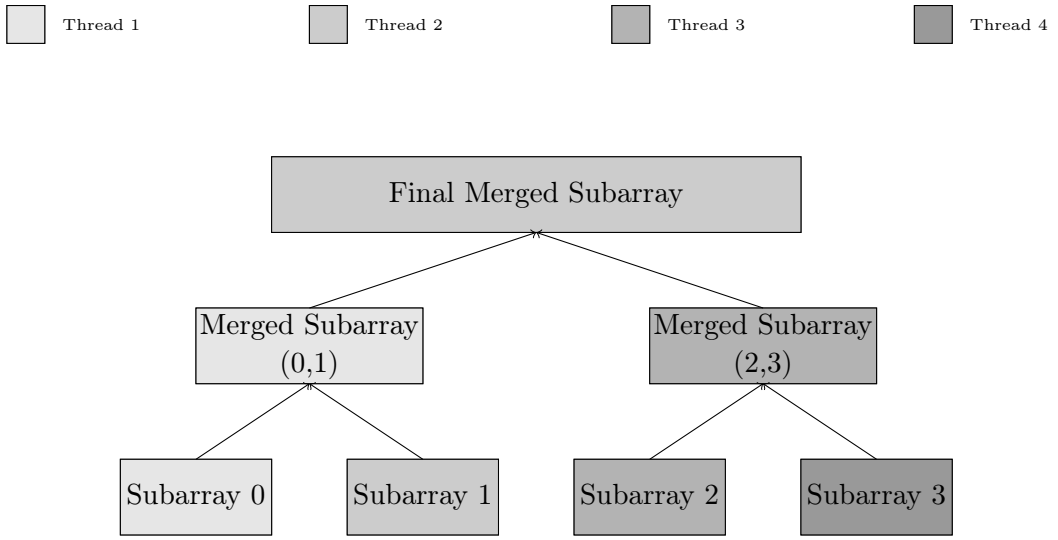


Figure 3.5. Parallelised Merge Sort with 4 Threads (Merging Step)

3.2.8 Nvidia Thrust

Nvidia Thrust is a parallel algorithms library that is part of the Nvidia CUDA toolkit that provides a collection of parallel algorithms for common tasks such as scanning, reducing or in our case sorting [64]. The main function that we utilise is the "thrust::sort_by_key" function which is used to sort the classifiers by their associated distances as an alternative to the parallelised merge sort algorithm that we implemented. The Thrust library implemented sort utilises the Radix Sort algorithm which is particularly performant when sorting large datasets of floating point values. One of the main concerns when using the library to implement sorting was the fact that if the sort operation was not stable, similar to Merge Sort [11], then we could not guaratee that the sort maintains the original ordering of elements that are equidistant from the query point. However, the Thrust library does provide a stable sort implementation which ensures that the original ordering of elements as Counting sort is used as a subroutine in the Radix Sort algorithm [64].

On the other hand, there is still present the problem that in the implementation of the KNN algorithm, the sorting of classifiers has to introduce a "tie-breaker rule" in cases where two or more classifiers are equidistant from the query point. This is particularly important as the KNN algorithm is a deterministic algorithm and should return the same results for the same input data. Some implementations of the KNN algorithm may introduce a random tie-breaker rule, however, this is not ideal as the generation of pseudo-random numbers can be a significant bottleneck in the algorithm. In addition, in the context of the size of our dataset and the number of dimensions, the probability of two classifiers being equidistant from the query point is quite low, hence in this implementation we simply choose the first classifier that is encountered in the sorted array.

3.2.9 Runtime Parameters

Running our KNN algorithm mini-application, we will make use of the Warwick University Department of Computer Science's HPC cluster, specifically the "Gecko" batch compute nodes. This is mainly due to the fact that they were amongst the few machines that had Nvidia A10 GPUs available for use and enough memory to fit our vast problem sizes. However, as we are limited by the fact that the user memory partition limit is 128GB [70], we had to limit the problem sizes to 100 million datapoints with 200 dimensions which will utilises 80GB for our N-Dimensional Space. The additional 48GB of memory are used as leeway for any intermediary calculations we may have to perform such as memory for sorting. The runtime parameters that we will be using for the KNN algorithm mini-application are as follows:

- Number of Datapoints: Variant
- Number of Dimensions: Variant
- K Number of Neighbours: 100
- Number of Query Points: 20
- Number of Classes: 10

The number of datapoints and dimensions are varied to test the scalability of the algorithm. in both growing dataset size and increasing dimensions. We also make full utilisation of compiler optimisations such as "-O3" and "-march=native" on the GCC g++ compiler [13] to ensure that the code is compiled with the highest level of optimisation which may increase compile times considerably but will improve the runtime performance of the algorithm when running tests. In addition within the runtime parameters of the remoterun.sh BASH script [22], we specify the following environment for the runtime of the KNN algorithm mini-application:

Version	Kokkos Optimised KNN
Partition	"Gecko"
CPU	AMD EPYC 7443
Core Count	24 Physical Cores, 48 Logical Threads
Total Host Memory	128GB
GPU	Nvidia A10 (24GB)
Compiler	g++ GCC 12.2
CUDA Toolkit	CUDA Devkit 12.3
Kokkos Version	4.1.00
Compiler Flags	-O3 -std=c++17 -funroll-loops -fopenmp-simd -march=core-avx2 -mtune=core-avx2 -mrtm -fopenmp
Test Repeats	Average of 5 Runs

Table 3.3: Runtime Parameters for Kokkos Optimised KNN Algorithm

3.3 Total Compute Time Results

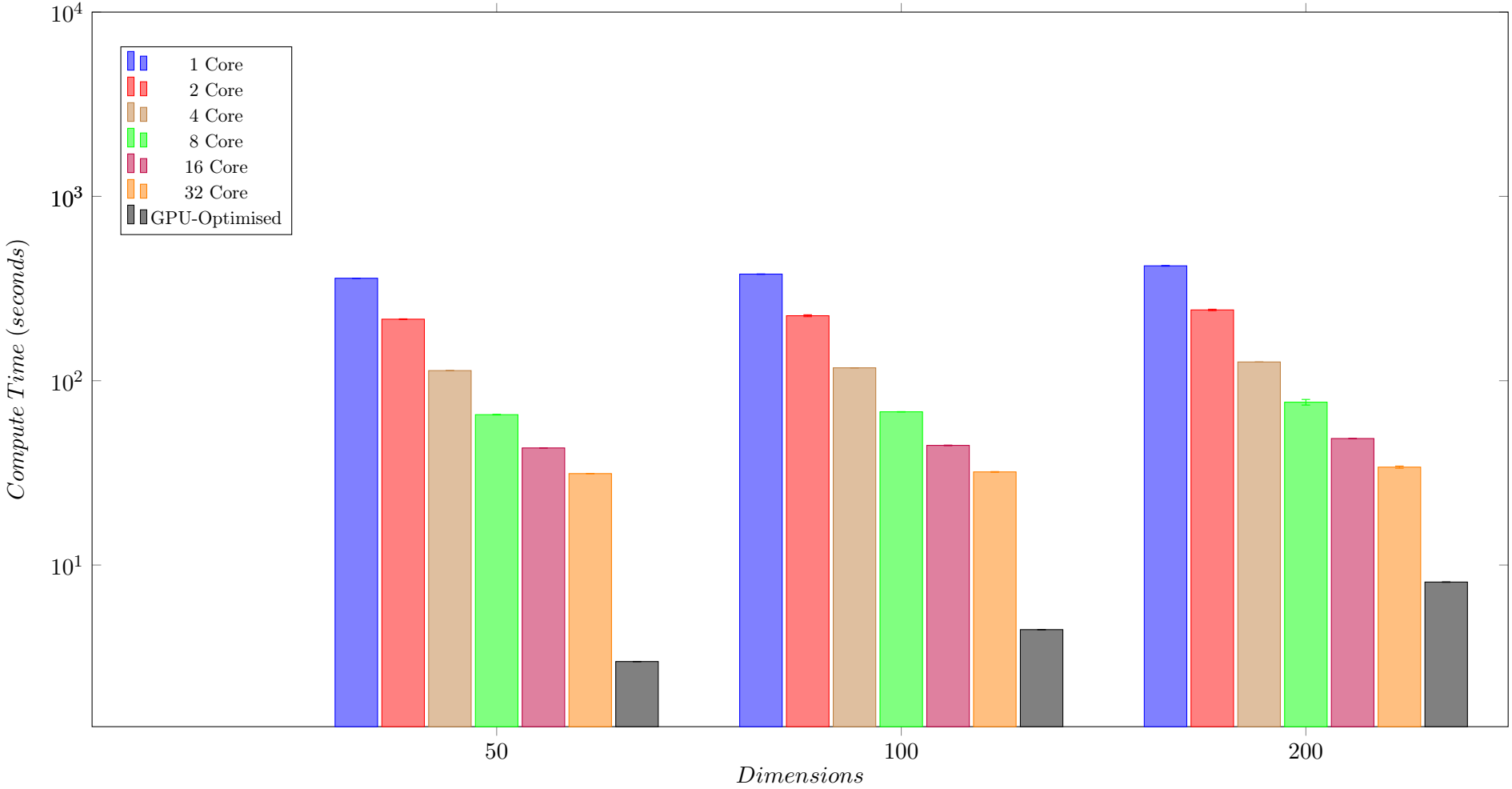


Figure 3.6. Compute Time vs Dimensions for Kokkos Optimised KNN Algorithm (100 Million)

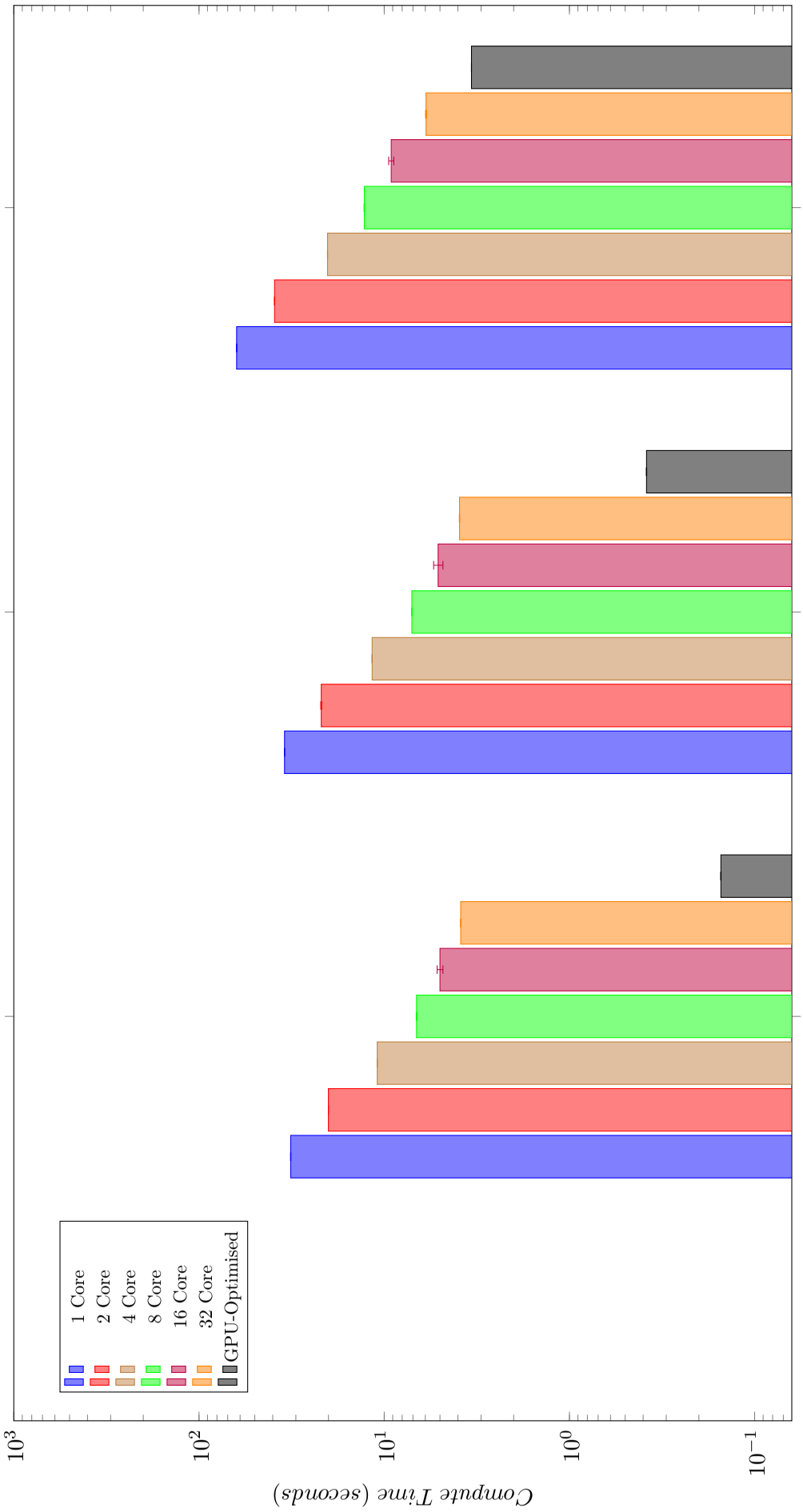


Figure 3.7. Compute Time vs Dimensions for Kokkos Optimised KNN Algorithm (10 Million)

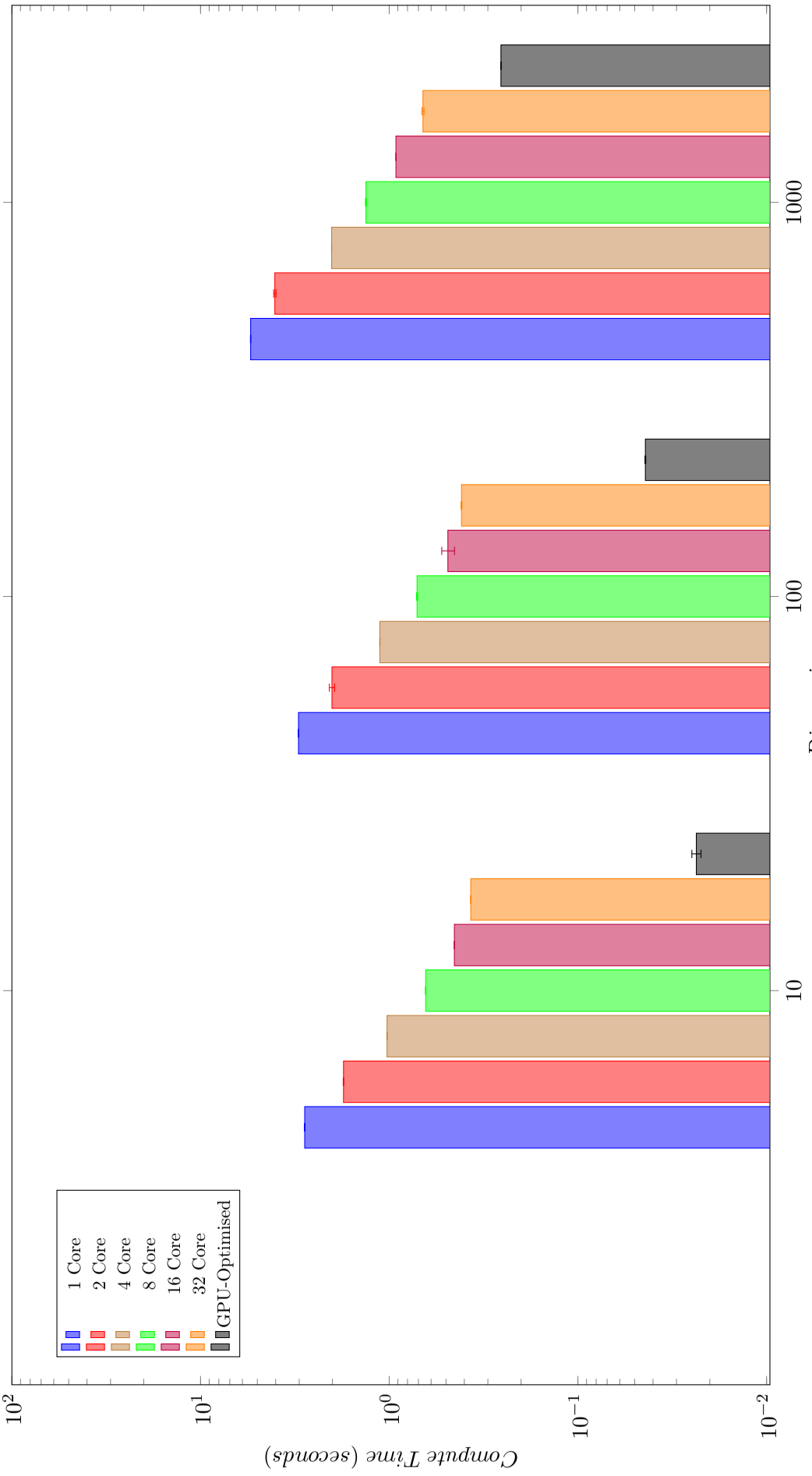


Figure 3.8. Compute Time vs Dimensions for Kokkos Optimised KNN Algorithm (1 Million)

3.4 Results Analysis

The results of the KNN optimised algorithm demonstrate the scalability of the algorithm with the use of increasing numbers of cores and massive parallelism through GPUs. The results show a linear relationship between the number of dimensions, the compute time and the core count. Important to note is the fact that the timings are the sum of total compute time for all query points, which is the combination of time used to calculate Euclidean distances, sorting and the final classification of the query points.

3.4.1 Euclidean Distance Analysis

We calculate the number of floating point operations (FLOPs) for the Euclidean distance calculation as follows:

$$\text{FLOPs} = 3 \times D \times N \times C \tag{3.2}$$

Where in this case, D is the number of dimensions, N is the number of datapoints and C is the number of query points. We use the convention that multiply and add operations count as separate instructions, hence the factor of 3 in the equation which accounts for calculating difference, squaring this difference and then the summation of squared differences. We utilise the FLOPs in order to calculate the theoretical peak performance of the algorithm which is the maximum number of floating point operations that can be performed in a given time. For reference the largest problem size that we run on the GPU is 100 million datapoints with 200 dimensions, we run the algorithm to query 20 points and classify them into 10 classes. Hence the FLOPs for this problem size is:

$$\text{FLOPs} = 3 \times 200 \times 100,000,000 \times 20 = 1.2 \text{ TFLOPS} \tag{3.3}$$

The theoretical peak performance of the Nvidia A10 GPU is 31.2 TFLOPS/s [62] on 32-bit floating point values whilst utilising Tensor Cores which are specialised hardware units that are able to perform matrix-matrix multiplication at a much faster rate. We do not utilise Tensor Cores in our implementation as the Kokkos library does not have support for Tensor Cores without the utilisation of Kokkos Kernels [75] which is a separate library that provides a collection of parallel algorithms for linear algebra operations. Our implementation of the KNN algorithm ran on this problem size achieved a time of 6.625559 seconds on average for the Euclidean distance calculation. We calculate the performance of the algorithm as follows:

$$\text{Performance} = \frac{\text{FLOPs}}{\text{Compute Time}} = \frac{1.2 \text{ TFLOPS}}{6.625559 \text{ seconds}} = 181.1 \text{ GFLOPS/s} \tag{3.4}$$

The disparity between the theoretical peak performance and the achieved performance can be explained by a number of factors such as memory bandwidth, memory latency, memory transfers and the overhead of the Kokkos library. In addition to this we do not utilise Tensor Cores present on the GPU. In fact the, the 32 Core CPU implementation of the algorithm is able to achieve a performance of 191.1 GFLOPS/s for the exact same problem size which is more performant than the GPU implementation. However, consider a problem size of 10 million datapoints with 100 dimensions, hence the N-Dimensional Space would be 4GB in size which is feasible to fit within the GPU memory. The GPU implementation of the algorithm is able to achieve a performance of 357.9 GFLOPS/s, whilst the 32 Core CPU implementation is able to achieve a performance of 175.8 GFLOPS/s. Clearly the application is bounded by the memory transfer speeds and the memory bandwidth of the GPU across the PCI-E bus.

3.4.2 Euclidean Distance Calculation Performance

We calculate the FLOPS/s performance of the Euclidean distance calculation for the Kokkos Optimised KNN Algorithm for problem sizes of 100 million datapoints and 10 million datapoints to guage the effect that memory transfer speed have on the performance of the algorithm.

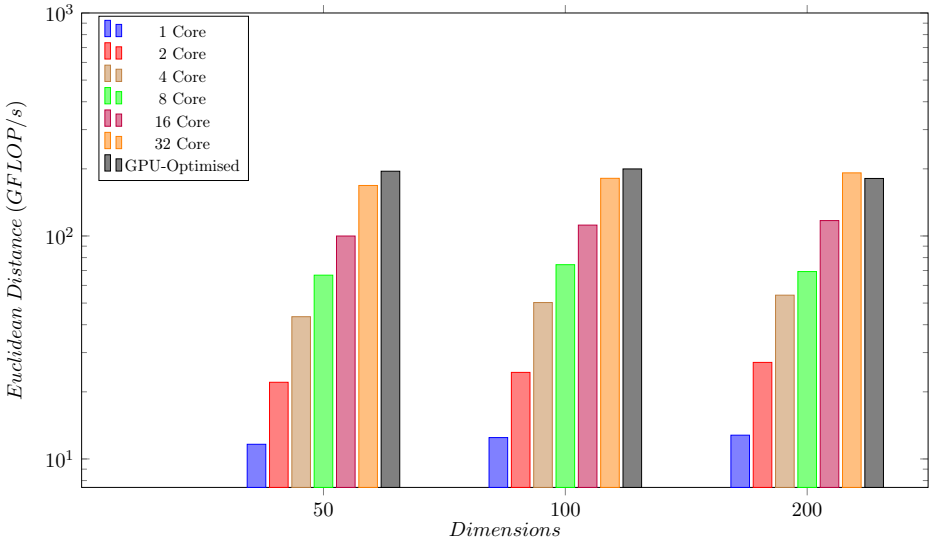


Figure 3.9. Euclidean Distance Performance vs Dimensions for Kokkos Optimised KNN Algorithm (100 Million)

The problem of the application being memory bound is further exemplified by the performance of the Euclidean distance Calculation when considering the GFLOP/s performance of the algorithm. As demonstrated in Figure 3.9, due to the cost of memory transfers between the host space and GPU, the performance of the GPU utilising our Kokkos Optimised Algorithm is halved compared to problem sizes that can fit within the GPU memory, such as in the case of Figure 3.10.

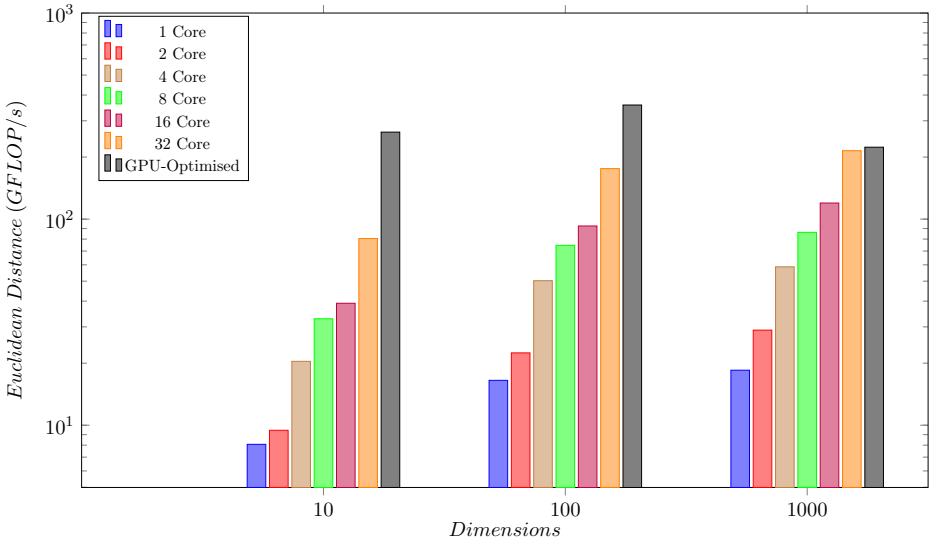


Figure 3.10. Euclidean Distance Performance vs Dimensions for Kokkos Optimised KNN Algorithm (10 Million)

We consider the bottleneck of the memory transfer speeds to be between the host memory and the GPU memory as the performance of the GPU tends to match that of the CPU multithreaded implementation when the problem size is quite large. We consider the memory bandwidth of the host memory of DDR4 DRAM running at 3200MHz to be 25.6GB/s per channel [16]. The AMD EPYC 7443 CPU has 8 memory channels which gives a total memory bandwidth of 204.8GB/s [4]. The Nvidia A10 GPU has a peak memory bandwidth of 600GB/s [62] which is significantly higher than the CPU. Therefore, the bottleneck in the GPU implementation of the algorithm is the maximum memory bandwidth of the host memory which largely explains why the performance of the Nvidia A10 drops to $\sim$ 190 GFLOPS/s when the workload partitioning scheme has to transfer data between the host memory and the GPU memory.

3.4.3 Parallelised Sorting Analysis

We observe the timings of the parallelised merge sort algorithm utilised in comparison to the Nvidia Thrust library. One can realise that due to differences in algorithmic time complexity, as merge sort is $O(n\log(n))$ [11] and Radix Sort is $O(n*c)$ (as Counting Sort is a subroutine of Radix Sort) [65], the performance of the Thrust library is significantly better than the parallelised merge sort algorithm. In addition, difference in hardware performance between the CPU and GPU is also a large factor in the performance of the sorting algorithm as the Nvidia A10 GPU has a higher peak memory bandwidth than the AMD EPYC 7443 CPU. Considering the fact that sorting is a memory bound operation due to the number of memory swaps that have to be made, as opposed to being compute bound, the performance of the sorting algorithm is largely determined by the memory bandwidth of the hardware.

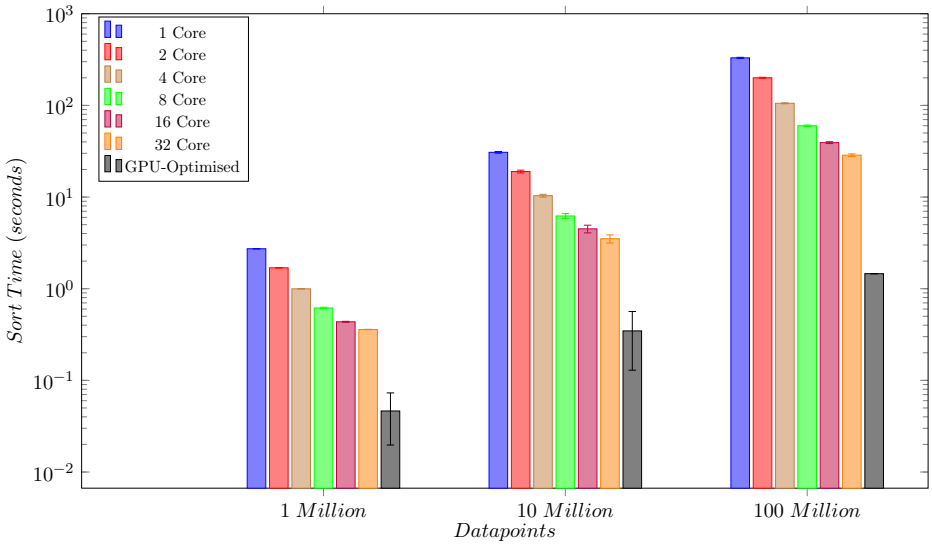


Figure 3.11. Sort Time vs Datapoints for Kokkos Optimised KNN Algorithm

Upon analysis that due to algorithmic complexity and differences in memory speed, that the performance of the sorting method with Nvidia Thrust is significantly better than the parallelised merge sort algorithm. However, in 3.11 it is found that the error bounds for the GPU-Optimised sorting algorithm are significantly larger clearly indicating that another factor is affecting the performance of the sort. We consider the fact that upon further investigation it was found that the dimensionality of the dataset was increasing the time taken to sort the dataset as demonstrated in Figure 3.12. An exception to this was for the problem size of 100 million datapoints as the performance was

asymptotically bounded by the number of elements that had to be sorted hence we only analyse the effect of dimensionality with 1 million and 10 million datapoints.

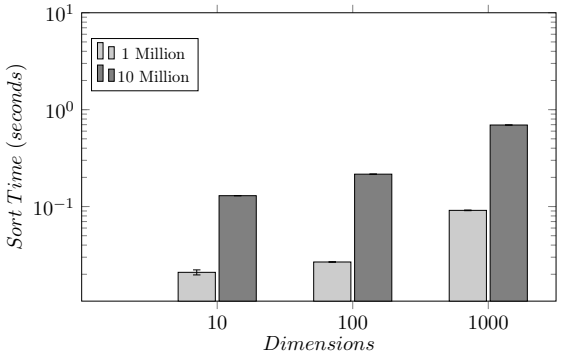


Figure 3.12. Sort Time vs Dimensions for Thrust Library SortByKey (1 Million & 10 Million datapoints)

The increase in time taken required to sort the dataset with increase in dimensionality can be explained by the fact that as Radix Sort is also asymptotically bounded by $O(n*c)$, where in n is the number of datapoints and c is the bit width size of the elements that are being sorted. As we increase dimensionality, the sizes of the euclidean distances increase which in turn increases the bit width of the elements that are being sorted. This is the reason why the time taken to sort the dataset increases with the dimensionality of the dataset. The merge sort algorithm is not affected by this problem as a result of the algorithm being comparison based [11] and not dependent on the elements being sorted.

3.5 Algorithm Validation

In order to validate the results of the Kokkos Optimised KNN algorithm, we compare the results of the algorithm from version to version utilising a checksum method. As the dataset is generated by a pseudo-random number generator, the results of the KNN algorithm should be deterministic for every run of the algorithm. That is, the output classifier for each query point should be the same from one run to another. Therefore, given the same input data, the algorithm is guaranteed to return the same results each time. This is vital in proving the correctness of the algorithm as in the case where in the checksums do not match, it would indicate that there is a flaw in the implementation.

```
1                    ||===============RESULTS==============||
2                    ||Data Point Set 0:            Class 8 ||
3                    ||Data Point Set 1:            Class 7 ||
4                    ||Data Point Set 2:            Class 6 ||
5                    ||Data Point Set 3:            Class 0 ||
6                    || ...                                 ||
7                    ||===============RESULTS==============||
```

Listing 3.9: Checksum Validation Method for Kokkos Optimised KNN Algorithm

The checksum validation method is a simple approach that involves comparing the output classes of the query points from one run of the algorithm to another. The results of the algorithm were outputted with the results for timings in the output results file. In turn, if any of the output classes were to change for the same problem size, it would indicate the presence of a race condition due to incorrect parallelisation or logical errors within the application.

3.6 Profiling Tools

For some of the smaller problem sizes, we utilise the C++ profiling tool, Cachegrind through the use of the Valgrind tool suite. Cachegrind is a cache profiler that simulates the behavious of the cache hierarchy to compute cache misses during the execution of our program [88].

```
==31805== Cachegrind, a cache and branch-prediction profiler
==31805== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==31805== Using Valgrind-3.21.0 and LibVEX; rerun with -h for copyright info
==31805== Command: ./knnGPUOptimised.host -N 100000 -D 10 -K 100 -NC 10 -C 20
==31805==
  User N is 100000
  User D is 10
  User K is 100
  User NC is 10
  User C is 20

============================================================================
Number of Threads is : 6.

||==========================RESULTS==========================
||Initialisation Time:                          0.185889
||N-Dimensional Euclid Time:                    0.637356
||Sort Time:                                    2.077117
||Total Compute Time:                           2.714473
||Total Time:                                   2.900362
||==========================RESULTS==========================
||Data Point Set 0:                              1
||Data Point Set 1:                              9
||Data Point Set 2:                              6
||Data Point Set 3:                              7
||Data Point Set 4:                              6
||Data Point Set 5:                              5
||Data Point Set 6:                              1
||Data Point Set 7:                              4
||Data Point Set 8:                              2
||Data Point Set 9:                              7
||Data Point Set 10:                             8
||Data Point Set 11:                             1
||Data Point Set 12:                             0
||Data Point Set 13:                             2
||Data Point Set 14:                             5
||Data Point Set 15:                             4
||Data Point Set 16:                             8
||Data Point Set 17:                             8
||Data Point Set 18:                             3
||Data Point Set 19:                             3
||==========================RESULTS==========================
==31805==
==31805== I refs:        1,520,464,816
[u2145461@viglab-53 knnGPUOptimised]$
```

Figure 3.13. Cachegrind Output for Kokkos Optimised KNN Algorithm

On the other hand, for the larger problem sizes, we utilise the Nvidia Nsight Compute tool, which supports GPUs. It allows for views of the GPU hardware counters and the performance of the GPU including extensive details about Warp utilisation, memory bandwidth, memory latency and the number of instructions executed by the GPU [59].

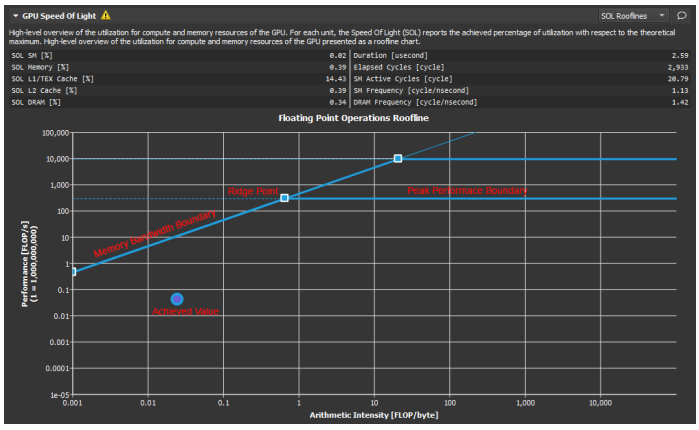


Figure 3.14. Sample Nvidia Nsight Compute Output Roofline Analysis [59]

4 Graph Convolutional Networks

Graph Neural Networks (GNNs) are a class of neural networks that were originally designed to apply neural network training to data that is inherently graph structured. For example, social networks, citation networks, biological networks and many more real world applications can have their data represented in the form of a graph structure [95]. The main idea behind Graph Convolutional Networks (GCNs) is to generalise the convolution operation from regular grid structures in the case of Convolutional Neural Networks (CNNs) to irregular graph structures [34]. We take Thomas Kipf's implementation of the GCN algorithm [34] and optimise it for performance.

Specifically, we aim to optimise GCNs in the context of node classification tasks on large scale graphs. The main aims of said optimisations were to reduce total memory usage by making use of highly efficient graph structures in Kokkos [19] and to improve performance of training epochs by parallelising the graph convolution, forward and backward propagation steps.

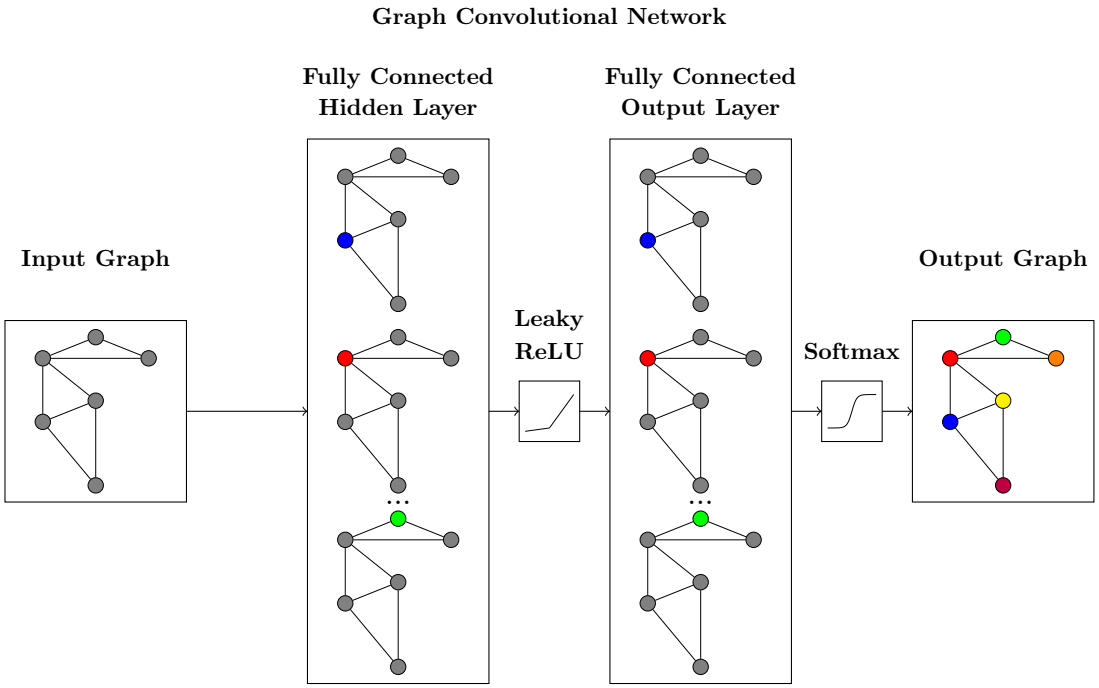


Figure 4.1. Graph Convolutional Network Architecture to be Optimised Adapted from Thomas Kipf's Paper [34]

We make use of this node properties classification architecture in order to take node information represented as features vectors. We then apply the convolution algorithm to take into account the node's neighbours and their features before training the network to classify the nodes into their respective classifiers.

GCNs were chosen as the target for performance portable optimisations as they are largely prolific in the field of Machine Learning and have a wide range of applications [95]. They are significantly more complex than the KNN algorithm in terms of algorithmic complexity [91]. Gradient Descent was used to train the network through forward and backward propagation steps. These steps are effectively sequences of matrix multiplications and element-wise operations and are the target operations for optimisation.

As a result, we develop two versions of the GCN algorithm, one where in the kernels are implemented entirely in the Kokkos ecosystem and another where in the kernels are implemented in CUDA C++ with the utilisation of highly optimised CUDA linear algebra libraries such as CUDA Basic Linear Subprograms (CuBLAS) [56] and CUDA Deep Neural Networks (CuDNN) [58]. The reasoning behind implementing the latter version is to guage the performance of Tensor Cores on the Nvidia A10 GPU compared to general purpose CUDA cores [62]. Additionally, we use it to guage the performance of library implemented functions compared to hand written kernels in Kokkos as a baseline comparison.

Finally, similar to the previous section with the KNN algorithm, the Kokkos optimised version will be written to be compiled for both multithreaded x86-64 CPU architectures and Nvidia GPU Architectures utilising the Kokkos performance portability ecosystem. Important to note is that as a result of increased experience with the Kokkos library, we make less use of "#ifdef" preprocessor directives and instead make use of the Kokkos execution space abstraction to write code that is performant across multiple architectures without having to explicitly rewrite large portions of the codebase for each architecture.

4.1 Design

The mathematics behind training GCNs with the Gradient Descent optimisation method is covered [76], specifically the linear algebra operations ustilised to perform backward and forward propagation. Additionally, we specify the convolution operations on graphs alongside the activation functions and our choice of loss function with their respective partial derivatives so we can propogate error with respect to our loss function through the network.

4.1.1 Graph Convolutions

The main idea behind Graph Convolutional Networks is to generalise the convolution operation from regular grid structures in the case of Convolutional Neural Networks (CNNs) to irregular graph structures [34]. There are many ways to define the graph convolution steps, for example we can utilise random walks, as given from the generalisation of graph convolutions from the GraphSAGE algorithm [25]. However, for our case we implement the GCN algorithm of taking the average of the features from a given node feature vector and it neighbours.

First we must define what a feature vector is for a given node. We consider the feature vector to be a vector of real numbers to represent the node's properties. For example, in the case of a social network, the feature vector could contain information such as the age of the person or the number of friends they have.
We then define the adjacency matrix of the graph to be a matrix of size $N \times N$ where in N is the number of nodes, and the element A_{ij} is 1 if there is an edge between node i and node j and 0 otherwise. As an example, we can consider the following undirected graph:

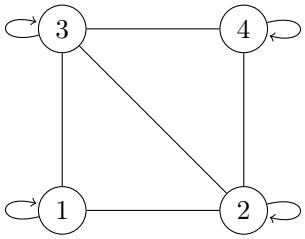


Figure 4.2. Example Undirected Graph

In the GCN paper, they add self loops to the adjacency matrix in order to account for the node's own features. This is effectively similar to adding the identity matrix to the adjacency matrix. We can represent the adjacency matrix of this graph as follows:

$$A = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{bmatrix} \tag{4.1}$$

We then define the degree matrix of the graph to be a diagonal matrix where in the element D_{ii} is the sum of the elements in the ith row of the adjacency matrix. We make utilisation of the degrees of the nodes in the graph to normalise the adjacency matrix. Therefore, the updated graph convolution will take the average of the feature vectors of the nodes and its neighbours. The degree matrix for the above graph is defined for A:

$$D = \begin{bmatrix} 3 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 3 \end{bmatrix} \tag{4.2}$$

Finally, we consider a feature matrix X where in the ith row of the matrix is the feature vector of the ith node. The feature matrix essentially contains the feature vectors of all the nodes in the graph. It effectively stores the node properties before we apply the graph convolution operation. We provide an arbitrary feature matrix for the above graph:

$$X = \begin{bmatrix} 0.7 & 0.5 \\ 0.0 & 1.0 \\ 0.9 & 0.2 \\ 0.3 & 0.3 \end{bmatrix} \tag{4.3}$$

Within the GCN algorithm, the original feature matrix can perform a convolution operation utilising the adjacency matrix, feature matrix and the inverse of the degree matrix. Hence, we can define the naive graph convolution step by taking the average of the feature vectors of the nodes and its neighbours as follows using matrix operations:

$$H = D^{-1}AX \tag{4.4}$$

On the other hand, there are issues with merely taking the average of the feature vectors neighbours. Namely, it was found that for large graphs in the case of "celebrity nodes", where in the node has a large number of neighbours, a singular node can dominate the feature vectors of its neighbours. To deal with this, we make use of the symmetric normalisation technique as defined in the GCN paper [34]. The symmetric normalisation trick ensures that when aggregating neighbouring nodes feature vectors, that the feature vectors are also scaled by the inverse square root of the nodes out degree. This ensures that in the case of "celebrity nodes", their weighting is normalised in regards to the number of neighbours they have. We define the normalisation trick:

$$\tilde{H} = D^{-\frac{1}{2}}AD^{-\frac{1}{2}}X \text{ [34]} \tag{4.5}$$

Note that due to the presence of inverse square roots utilised in the normalisation trick, the implementation can become quite computationally expensive. Therefore, we try to balance the trade off between computational complexity and the performance of the model by considering different depths of neighbourhood aggregation.

4.1.2 Increasing Neighbourhood Depth

In the case of the GCN algorithm, we can increase the depth of the neighbourhood that we consider when aggregating the feature vectors of the nodes. We implement the following simple iterative definition to aggregate deeper and deeper neighbourhoods of the graph:

$$H^{(l+1)} = D^{-\frac{1}{2}} A D^{-\frac{1}{2}} H^{(l)} \tag{4.6}$$

Where in $H^{(l)}$ is the feature matrix of the lth layer of the graph convolutional network and $H^{(l)}$ is the feature matrix of the lth layer of the graph. The rationale as to why we require deeper graph convolutions is that in the case of large graphs, we may want to consider the feature vectors of nodes that are further away from the node. For example in the context of a social network, we may want to consider the friends of friends of a person in order to classify them into a certain category [95].

The methodology as to how this definition calculates deeper neighbourhood node aggregation comes from the fact that by properties of transitivity. That is if node i aggregates node information from its immediate 1-hop neighbourhood, then nodes within the 1-hop neighbourhood of i also aggregate information from their 1-hop neighbourhood. Therefore, by iterating this process and performing the aggregation function arbitrarily many times, we can consider the feature vectors of nodes that are further away from the node.

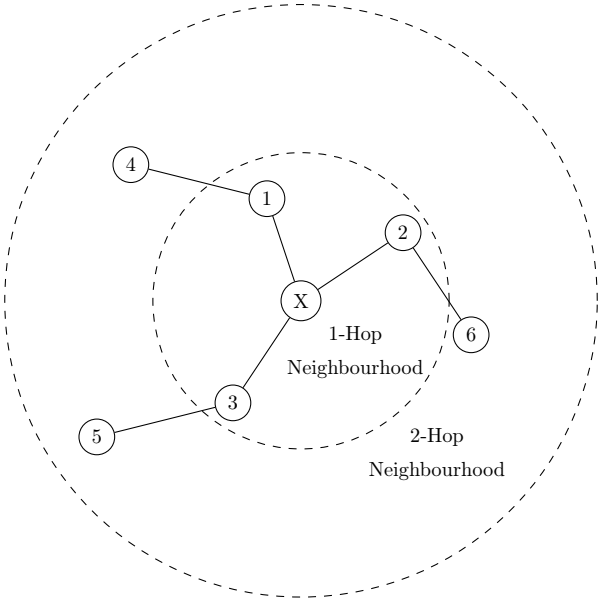


Figure 4.3. Increasing Neighbourhood Depth in Graph Convolutional Networks [25]

The above figure demonstrates the concept of deeper node graph convolutions that aggregate neighbour information from further away nodes. The dashed circles represent the 1-hop and 2-hop neighbourhoods of node X respectively. However, there whilst aggregating deeper neighbourhoods can be beneficial as we effectively aggregate more information from the graph, it can also be computationally expensive [17].

Additionally, deeper node aggregation can lead to the problem of over smoothing in the case of node classification tasks. Over smoothing is the phenomenon where in the feature vectors for each of the nodes in the graph become increasingly similar as we perform deeper node aggregation [77]. This can lead to the loss of important node information and can lead to the network classifying all nodes in the graph to be the same category. In turn, many papers have found that the optimal neighbourhood depth for node classification tasks is around 2-4 hops [25][33].

4.1.3 Neural Network Architecture

The architecture of the neural network then takes in the feature matrix of the nodes after performing the graph convolution operation. We then apply the training iterations on these "node embeddings" in order to classify the nodes into their respective categories. The forward propagation algorithm utilises non-linear activation functions and fully connected network layers to classify the nodes [92].

We consider a full connected layer to be a matrix multiplication operation followed by element wise operations of the addition of a bias vector. Given a feature matrix H of the nodes, passing these feature vectors through a fully connected layer with weights W^1 and bias b^1 where in the dimensions of the Weights and Bias vectors match the dimensions of the feature vectors of the nodes by the size of the input to the next hidden layer. To propogate the feature vectors through fully connected layer is the operation:

$$O^1 = HW^1 + b^1 \tag{4.7}$$

Where in O^1 is the output of the fully connected layer. We can the optionally apply a non-linear activation function dependent on the need of the output to the next layer of the neural network.

4.1.4 Activation Functions

The main non-linear activation function that we consider are the Rectified Linear Unit (ReLU) and the Softmax activation function. The ReLU activation function is utilised heavily in the field of deep learning due to the simplicity of implementation, fast compute time and the fact that it does not suffer from the vanishing gradient problem for inputs greater than zero [92].

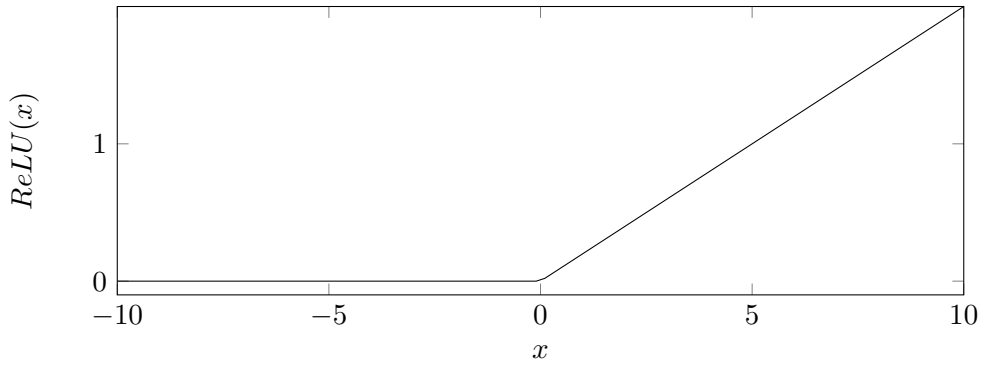


Figure 4.4. ReLU Activation Function

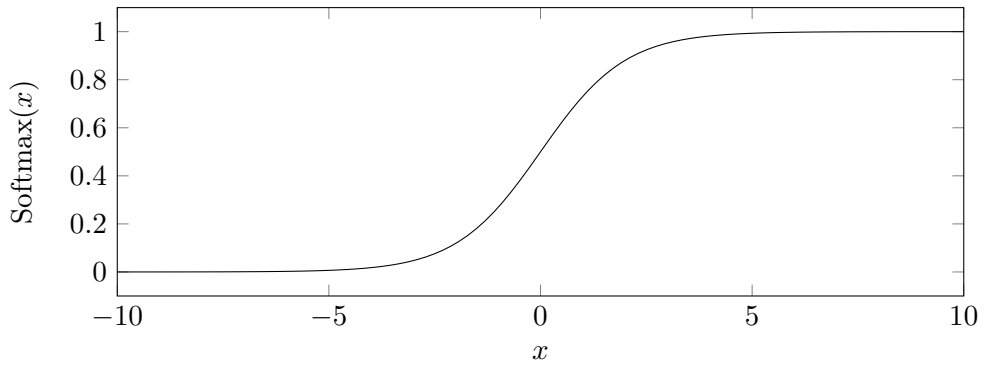


Figure 4.5. Softmax Activation Function

We use the standard definition of the ReLU activation function:

$$\text{ReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases} \tag{4.8}$$

Although, ReLU is ubiquitous in the field of deep learning, it does have the issue of the "dying ReLU" problem. Where in the gradient of the ReLU function is zero for inputs less than zero. This can cause the weights of the neural network to not update and stop learning [93]. In turn, we consider alternative forms of the ReLU activation function such as the Leaky ReLU function. that still allows for the gradient to be non-zero for inputs less than zero. Given an arbitrary scaling factor α, the Leaky ReLU function is defined as:

$$\text{LeakyReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{otherwise} \end{cases} \tag{4.9}$$

The Softmax activation function was used to output the probability distribution of the classes of the nodes on the output layer. The Softmax function normalises the output of the final fully connected layer and is often used in conjunction with a technique known as one hot encoding in order to train neural networks for classification tasks [26]. Important to note, we normalise the Softmax input by subtracting the maximum value of the input vector in order to prevent numerical instability [92].

$$\text{Softmax}(x_i) = \frac{e^{x_i}}{\sum_j^C e^{x_j}} \tag{4.10}$$

Where in C is the number of possible classifiers, x_i is the ith element of the output of the final fully connected layer and j is the input exponentiated value from the other output nodes. Consider the case where in we have 7 output classes for a given node and we apply the Softmax function to the output of the final fully connected layer W.

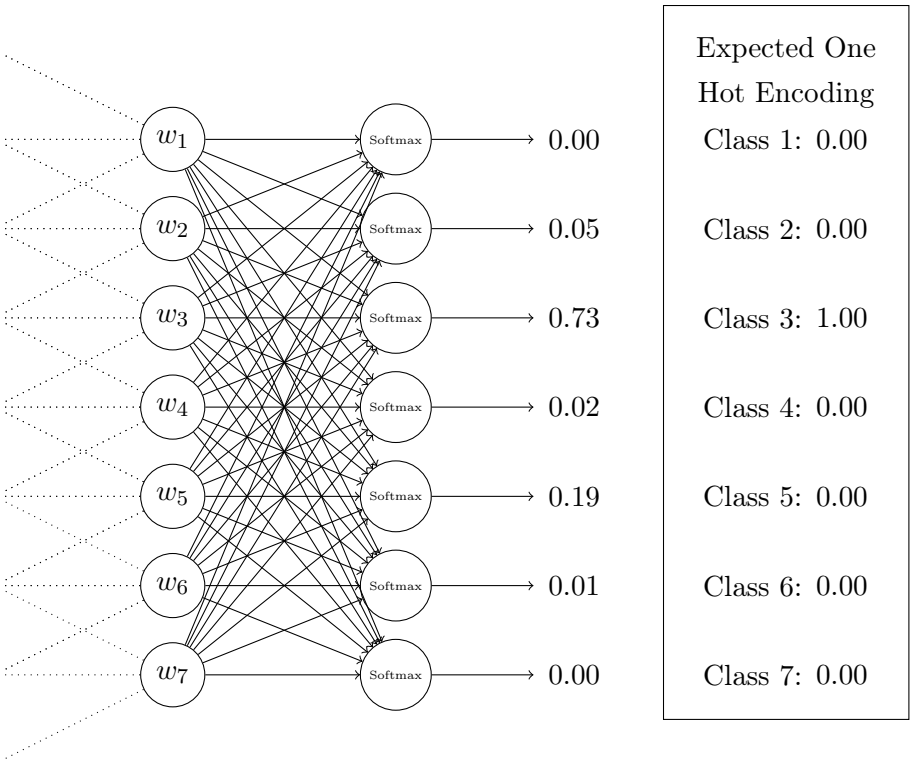


Figure 4.6. Softmax Activation Function Output for a Given Node with Ground Truth

Note that the output of the Softmax functions are the probabilities that the node belongs to a certain classifier. In the case of the above example, the node has a 73% probability of being classified as class 3, hence we select class 3 as the output classifier for the node.

4.1.5 Forward Propagation Rule

The definition of our forward propagation rule, given the updated node embeddings after our aggregation function, relevant activation functions and the definition of fully connected layers, describes the process of forward propagating the node embeddings through the neural network to classify the nodes into their respective categories. We provide a two layer neural network for the task of node property prediction. Given the node embeddings H after the graph convolution operation:

$$\text{Forward}\left(H\right) = \text{Softmax}\left(\text{LeakyReLU}\left(HW^1 + b^1\right)W^2 + b^2\right) \tag{4.11}$$

Where in W^1 and W^2 are the randomly initialised weight matrices of the fully connected layers, b^1 and b^2 are the bias vectors of the fully connected layers.

4.1.6 Full Batch Gradient Descent & Backward Propagation

Gradient Descent is an optimisation algorithm that is commonly used for unbounded optimisation problems [76]. The algorithm works iteratively by updating values of the weights and biases matrices with respect to a predefined loss function [30]. When considering full batch gradient descent, it performs training epochs by calculating the gradients over the entire dataset. Effectively, we forward propagate the entire dataset of node embeddings and backpropagate the error over the entire output of the neural network. The loss function is utilised as a measure of the error generated by the output of the neural network with respect to the ground truth labels of the training data [94].

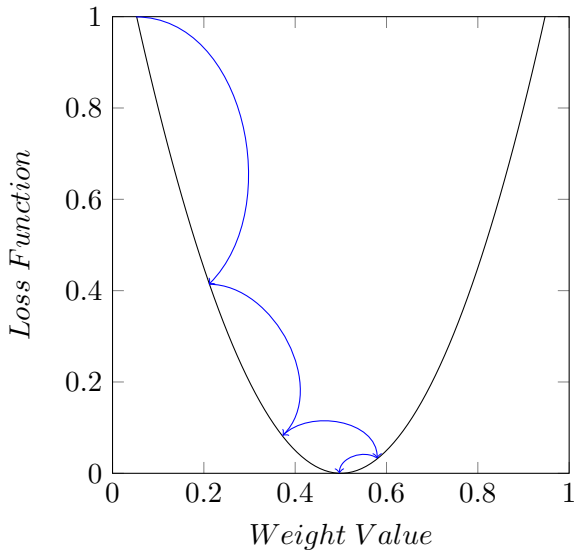


Figure 4.7. Visualised Gradient Descent Algorithm to Minimise the Loss Function

In our case, as we are using the Softmax activation function on the output layer, we use the Cross Entropy loss function. The main rationale behind doing so is that combined with 1-hot encoding of the ground truth labels, the Cross Entropy loss function combined with the Softmax activation function makes the calculation of gradients with respect to the loss function simpler [37].

In order to calculate the Cross Entropy Loss function, we consider the outputs of our Softmax function at the final layer of our graph neural network. Given the output of the Softmax function S and the ground truth labels Y, the Cross Entropy loss function is defined as:

$$\text{CrossEntropy}(\hat{Y}, Y) = -\sum_{i}^{C} \hat{Y}_i \log\left(Y_i\right) \tag{4.12}$$

On the other hand, considering the computational complexity of calculating logarithms we must find an alternative way to calculate gradients with respect to the loss function. In fact, we can entirely avoid the calculation of the logarithm by making use of a different loss function to measure loss per training epoch, however we still use derivative of Cross Entropy loss with respect to the derivative of the Softmax function when calculating gradients. To measure the loss per training epoch, we use the Mean Squared Error (MSE) loss function. Given the output of the Softmax function $\hat{Y}$ and the ground truth labels Y, we define the MSE loss function as:

$$\text{MSE}(\hat{Y}, Y) = \frac{1}{C}\sum_{i}^{C}\left(\hat{Y}_i - Y_i\right)^2 \tag{4.13}$$

4.1.7 Backward Propagation Rule

We consider the following example of the backward propagation rule for the two layer neural network defined in the forward propagation rule (4.11).

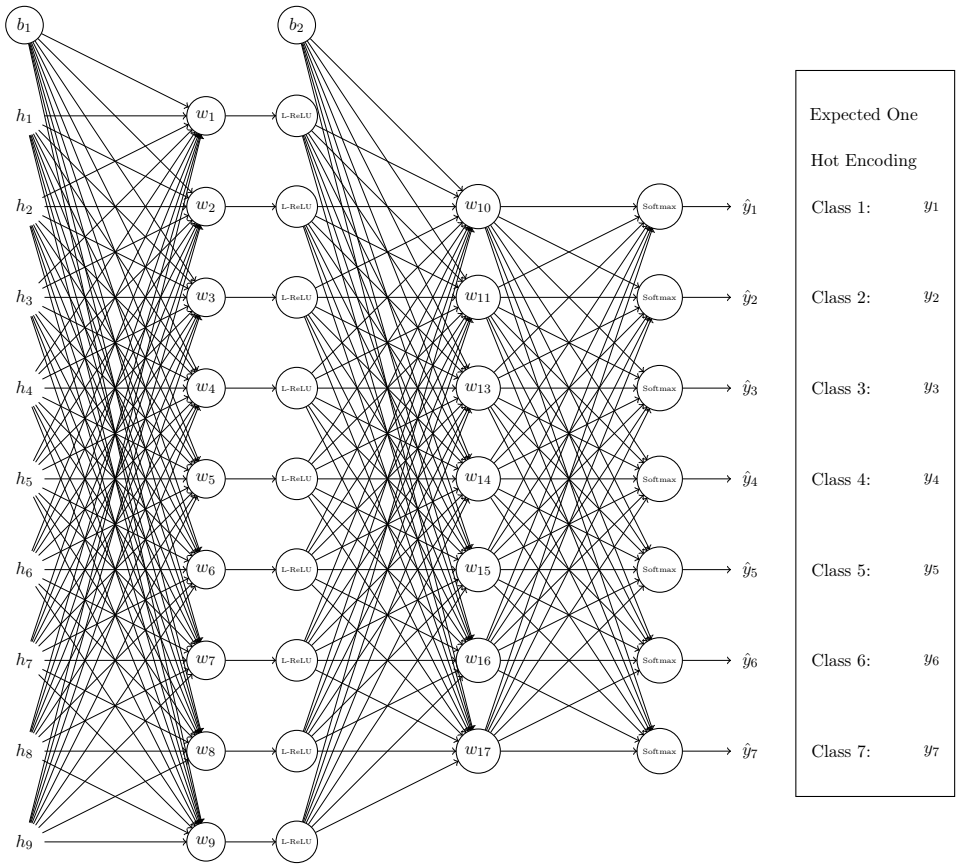


Figure 4.8. 9-Input 7-Class Neural Network defined in Forward Propagation Rule (4.11)

We provide an example of backpropagation on the network given in Figure 4.8 where in this case we want to update the weight of $w_1[1]$ within the first hidden layer with respect to the loss function errors generated by the output at the final layer by $\hat{y}_1$. This is where in $w_1[1]$ is the weight of the first input node h_1 in the first hidden layer node: w_1. We consider the path from $w_1[1]$ to $\hat{y}_1$ needed to calculate the errors and backpropagate the errors with respect to the derivatives along the path. The path from $w_1[1]$ to $\hat{y}_1$ can be defined ignoring the bias nodes as the derivative of a constant is zero. The path from $w_1[1]$ to $\hat{y}_1$ is defined as:

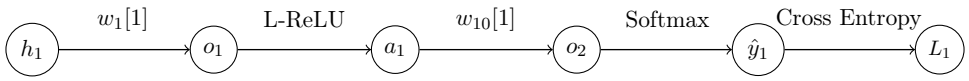


Figure 4.9. Path from $w_1[1]$ to $\hat{y}_1$ in the 9-Input 7-Class Neural Network

Where in o_1 and o_2 are the pre-activation outputs of the layer and a_2 and $\hat{y}_1$ are the post-activation outputs of a layer. In turn we can define the derivative of the loss L_1 at $\hat{y}_1$ in terms of the weight $w_1[1]$ by making use of the chain rule considering the path in Figure 4.9. In turn the gradient $\frac{\partial L_1}{\partial w_1[1]}$ can be found by chaining partial derivatives along the path:

$$\frac{\partial L_1}{\partial w_1[1]} = \frac{\partial L_1}{\partial \hat{y}_1} \cdot \frac{\partial \hat{y}_1}{\partial o_2} \cdot \frac{\partial o_2}{\partial a_1} \cdot \frac{\partial a_1}{\partial o_1} \cdot \frac{\partial o_1}{\partial w_1[1]} \tag{4.14}$$

We define the loss function as Cross Entropy loss function L and we define the derivative of the loss function with respect to the output of the Softmax function as:

$$\frac{\partial L(\hat{y}, y)}{\partial o_2} = \hat{y} - y \tag{4.15}$$

This is a result that can be derived by taking the derivative of the Cross Entropy loss function by the derivative of the Softmax function [37]. Note that we do not have to explicitly consider the derivatives of Softmax or Cross Entropy loss functions in the backpropagation rule as the result in Equation (4.15) accounts for both. Given that $\frac{\partial o_2}{\partial a_1} = w_{10}[1]$ as extrapolated from Figure 4.9, we then have to consider the derivative of the Leaky ReLU function with respect to the pre-activation output o_1 from $\frac{a_1}{o_1}$. Given a scalar constant α for the Leaky ReLU function, the derivative can be found:

$$\frac{\partial a_1}{\partial o_1} = \begin{cases} 1 & \text{if } o_1 > 0 \\ \alpha & \text{if } o_1 \leq 0 \end{cases} \tag{4.16}$$

The final derivative to consider is the derivative of the pre-activation output o_1 with respect to the weight $w_1[1]$ which is merely the input h_1 to the weight $w_1[1]$. Therefore in order to calculate the gradient $\frac{\partial L_1}{\partial w_1[1]}$ we can substitute the derivatives in Equations (4.15), (4.16) and (4.17) into Equation (4.14) to find the gradient $\frac{\partial L_1}{\partial w_1[1]}$:

$$\frac{\partial L_1}{\partial w_1[1]} = (\hat{y}_1 - y_1) \cdot w_{10}[1] \cdot \begin{cases} 1 & \text{if } o_1 > 0 \\ \alpha & \text{if } o_1 \leq 0 \end{cases} \cdot h_1 \tag{4.17}$$

The gradient $\frac{\partial L_1}{\partial w_1[1]}$ can be used to update the weight $w_1[1]$ in the first hidden layer with a given learning rate η. The weight update rule for the weight $w_1[1]$ can be defined as:

$$w_1[1] = w_1[1] - \eta \cdot \frac{\partial L_1}{\partial w_1[1]} \tag{4.18}$$

The weight update rule in Equation (4.18) can be applied to all the weights in the neural network with respect to all the gradients or errors calculated by the loss function. The generalised matrix operations to update the weights in the hidden layer of the neural network can be defined as W^1 with $(\cdot)^T$ denoting the transpose of the matrix:

$$\left(W^1\right)^T = \left(W^1\right)^T - \eta \cdot \left(\left(\hat{Y} - Y\right) \cdot W^2\right)^T \cdot \partial LReLU(O^1) \cdot \left(H^1\right)^T \tag{4.19}$$

The function applied to the pre-activation output O^1 in the hidden layer is the application of finding the derivative of the Leaky ReLU function on the pre-activation output O^1 which is defined as $\partial LReLU(O^1)$ in (4.19). We provide a function procedure for calculating $\partial LReLU(O^1)$:

Algorithm 3 Leaky ReLU Derivative

```
1: function LeakyReLU(O^1)
2:     for o ∈ O^1 do
3:         if o > 0 then
4:             ∂LReLU(o) ← 1
5:         else
6:             ∂LReLU(o) ← α
7:         end if
8:     end for
9:     return ∂LReLU(O^1)
10: end function
```

We can apply the same techniques to define the weights and biases rules for W^2, b^1 and b^2 in the neural network. Considering that the path from the output at the loss function is closer to the output layer, the gradients and error calculations are more straightforward to calculate. The weights within W^2 can therefore be updated with a similar weight update rule as defined in Equation (4.18). On the other hand we only consider inputs to the final output layer and ignore the inputs from the node embeddings at H. The weight update rule for W^2:

$$\left(W^2\right)^T = (W^2)^T - \eta \cdot \left(\hat{Y} - Y\right)^T \cdot \left(A^1\right)^T \tag{4.20}$$

The bias update rule for b^1 and b^2 can be found by considering the gradients of the loss function with respect to the biases. For the bias update rule for b^1 and b^2 we can define the bias update rule as:

$$b^1 = b^1 - \eta \cdot \left(\left(\hat{Y} - Y\right) \cdot W^2\right)^T \cdot \partial LReLU(O^1) \tag{4.21}$$

Similarly the bias update rule for b^2 we have the matrix operations:

$$b^2 = b^2 - \eta \cdot \left(\hat{Y} - Y\right)^T \tag{4.22}$$

The backpropagation rule defined in this section can be applied to the neural network defined in the forward propagation rule in Section 4.2 in order to minimise the output of the loss function. The rule is used within the training epochs of the neural network to update the weights and biases in the neural network after an instance of forward propagation. Backpropagation is one of the main function we aim to optimise in the neural network in order to reduce overall training time.

4.1.8 The Need for Fast Matrix Transposes

In the case of the backpropagation rules defined in Section (4.1.6), we can see that the matrix operations undergo a transpose operation before updating the weights and biases in the neural network. Matrix transposes are extremely common in training neural networks due to the fact that in many cases as a result of optimising memory access pattern for both the CPU and GPU that the memory access pattern for transposing a matrix is significantly faster than the memory access pattern for the original matrix [19].

$$\begin{bmatrix} v_{1,1} & v_{1,2} & \cdots & v_{1,n} \\ v_{2,1} & v_{2,2} & \cdots & v_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ v_{m,1} & v_{m,2} & \cdots & v_{m,n} \end{bmatrix}^T = \begin{bmatrix} v_{1,1} & v_{2,1} & \cdots & v_{m,1} \\ v_{1,2} & v_{2,2} & \cdots & v_{m,2} \\ \vdots & \vdots & \ddots & \vdots \\ v_{1,n} & v_{2,n} & \cdots & v_{m,n} \end{bmatrix} \tag{4.23}$$

The memory access pattern for the weights update can be visualised as a 2-dimensional reduction sum where each element in the weights matrix must be updated by the dot products of the gradients and the associated input values to the previous layer. Therefore to access the elements in both matrices efficiently, a contiguous memory access pattern is preferred necessitating the need for a transpose to improve concepts of memory and spatial locality to make more effective use of cache memory [84].

$w_{(0,0)}$	$w_{(1,0)}$	$w_{(2,0)}$	$w_{(3,0)}$	$w_{(4,0)}$	$w_{(5,0)}$	$w_{(6,0)}$	$w_{(7,0)}$	$w_{(8,0)}$	$w_{(9,0)}$
$w_{(0,1)}$	$w_{(1,1)}$	$w_{(2,1)}$	$w_{(3,1)}$	$w_{(4,1)}$	$w_{(5,1)}$	$w_{(6,1)}$	$w_{(7,1)}$	$w_{(8,1)}$	$w_{(9,1)}$
$w_{(0,2)}$	$w_{(1,2)}$	$w_{(2,2)}$	$w_{(3,2)}$	$w_{(4,2)}$	$w_{(5,2)}$	$w_{(6,2)}$	$w_{(7,2)}$	$w_{(8,2)}$	$w_{(9,2)}$
$w_{(0,3)}$	$w_{(1,3)}$	$w_{(2,3)}$	$w_{(3,3)}$	$w_{(4,3)}$	$w_{(5,3)}$	$w_{(6,3)}$	$w_{(7,3)}$	$w_{(8,3)}$	$w_{(9,3)}$
$w_{(0,4)}$	$w_{(1,4)}$	$w_{(2,4)}$	$w_{(3,4)}$	$w_{(4,4)}$	$w_{(5,4)}$	$w_{(6,4)}$	$w_{(7,4)}$	$w_{(8,4)}$	$w_{(9,4)}$
$w_{(0,5)}$	$w_{(1,5)}$	$w_{(2,5)}$	$w_{(3,5)}$	$w_{(4,5)}$	$w_{(5,5)}$	$w_{(6,5)}$	$w_{(7,5)}$	$w_{(8,5)}$	$w_{(9,5)}$
$w_{(0,6)}$	$w_{(1,6)}$	$w_{(2,6)}$	$w_{(3,6)}$	$w_{(4,6)}$	$w_{(5,6)}$	$w_{(6,6)}$	$w_{(7,6)}$	$w_{(8,6)}$	$w_{(9,6)}$
$w_{(0,7)}$	$w_{(1,7)}$	$w_{(2,7)}$	$w_{(3,7)}$	$w_{(4,7)}$	$w_{(5,7)}$	$w_{(6,7)}$	$w_{(7,7)}$	$w_{(8,7)}$	$w_{(9,7)}$
$w_{(0,8)}$	$w_{(1,8)}$	$w_{(2,8)}$	$w_{(3,8)}$	$w_{(4,8)}$	$w_{(5,8)}$	$w_{(6,8)}$	$w_{(7,8)}$	$w_{(8,8)}$	$w_{(9,8)}$
$w_{(0,9)}$	$w_{(1,9)}$	$w_{(2,9)}$	$w_{(3,9)}$	$w_{(4,9)}$	$w_{(5,9)}$	$w_{(6,9)}$	$w_{(7,9)}$	$w_{(8,9)}$	$w_{(9,9)}$

$g_{(0,0)}$	$g_{(1,0)}$	$g_{(2,0)}$	$g_{(3,0)}$	$g_{(4,0)}$	$g_{(5,0)}$	$g_{(6,0)}$	$g_{(7,0)}$	$g_{(8,0)}$	$g_{(9,0)}$
$g_{(0,1)}$	$g_{(1,1)}$	$g_{(2,1)}$	$g_{(3,1)}$	$g_{(4,1)}$	$g_{(5,1)}$	$g_{(6,1)}$	$g_{(7,1)}$	$g_{(8,1)}$	$g_{(9,1)}$
$g_{(0,2)}$	$g_{(1,2)}$	$g_{(2,2)}$	$g_{(3,2)}$	$g_{(4,2)}$	$g_{(5,2)}$	$g_{(6,2)}$	$g_{(7,2)}$	$g_{(8,2)}$	$g_{(9,2)}$
$g_{(0,3)}$	$g_{(1,3)}$	$g_{(2,3)}$	$g_{(3,3)}$	$g_{(4,3)}$	$g_{(5,3)}$	$g_{(6,3)}$	$g_{(7,3)}$	$g_{(8,3)}$	$g_{(9,3)}$
$g_{(0,4)}$	$g_{(1,4)}$	$g_{(2,4)}$	$g_{(3,4)}$	$g_{(4,4)}$	$g_{(5,4)}$	$g_{(6,4)}$	$g_{(7,4)}$	$g_{(8,4)}$	$g_{(9,4)}$
$g_{(0,5)}$	$g_{(1,5)}$	$g_{(2,5)}$	$g_{(3,5)}$	$g_{(4,5)}$	$g_{(5,5)}$	$g_{(6,5)}$	$g_{(7,5)}$	$g_{(8,5)}$	$g_{(9,5)}$
$g_{(0,6)}$	$g_{(1,6)}$	$g_{(2,6)}$	$g_{(3,6)}$	$g_{(4,6)}$	$g_{(5,6)}$	$g_{(6,6)}$	$g_{(7,6)}$	$g_{(8,6)}$	$g_{(9,6)}$
$g_{(0,7)}$	$g_{(1,7)}$	$g_{(2,7)}$	$g_{(3,7)}$	$g_{(4,7)}$	$g_{(5,7)}$	$g_{(6,7)}$	$g_{(7,7)}$	$g_{(8,7)}$	$g_{(9,7)}$
$g_{(0,8)}$	$g_{(1,8)}$	$g_{(2,8)}$	$g_{(3,8)}$	$g_{(4,8)}$	$g_{(5,8)}$	$g_{(6,8)}$	$g_{(7,8)}$	$g_{(8,8)}$	$g_{(9,8)}$
$g_{(0,9)}$	$g_{(1,9)}$	$g_{(2,9)}$	$g_{(3,9)}$	$g_{(4,9)}$	$g_{(5,9)}$	$g_{(6,9)}$	$g_{(7,9)}$	$g_{(8,9)}$	$g_{(9,9)}$

$a_{(0,0)}$	$a_{(0,1)}$	$a_{(0,2)}$	$a_{(0,3)}$	$a_{(0,4)}$	$a_{(0,5)}$	$a_{(0,6)}$	$a_{(0,7)}$	$a_{(0,8)}$	$a_{(0,9)}$
$a_{(1,0)}$	$a_{(1,1)}$	$a_{(1,2)}$	$a_{(1,3)}$	$a_{(1,4)}$	$a_{(1,5)}$	$a_{(1,6)}$	$a_{(1,7)}$	$a_{(1,8)}$	$a_{(1,9)}$
$a_{(2,0)}$	$a_{(2,1)}$	$a_{(2,2)}$	$a_{(2,3)}$	$a_{(2,4)}$	$a_{(2,5)}$	$a_{(2,6)}$	$a_{(2,7)}$	$a_{(2,8)}$	$a_{(2,9)}$
$a_{(3,0)}$	$a_{(3,1)}$	$a_{(3,2)}$	$a_{(3,3)}$	$a_{(3,4)}$	$a_{(3,5)}$	$a_{(3,6)}$	$a_{(3,7)}$	$a_{(3,8)}$	$a_{(3,9)}$
$a_{(4,0)}$	$a_{(4,1)}$	$a_{(4,2)}$	$a_{(4,3)}$	$a_{(4,4)}$	$a_{(4,5)}$	$a_{(4,6)}$	$a_{(4,7)}$	$a_{(4,8)}$	$a_{(4,9)}$
$a_{(5,0)}$	$a_{(5,1)}$	$a_{(5,2)}$	$a_{(5,3)}$	$a_{(5,4)}$	$a_{(5,5)}$	$a_{(5,6)}$	$a_{(5,7)}$	$a_{(5,8)}$	$a_{(5,9)}$
$a_{(6,0)}$	$a_{(6,1)}$	$a_{(6,2)}$	$a_{(6,3)}$	$a_{(6,4)}$	$a_{(6,5)}$	$a_{(6,6)}$	$a_{(6,7)}$	$a_{(6,8)}$	$a_{(6,9)}$
$a_{(7,0)}$	$a_{(7,1)}$	$a_{(7,2)}$	$a_{(7,3)}$	$a_{(7,4)}$	$a_{(7,5)}$	$a_{(7,6)}$	$a_{(7,7)}$	$a_{(7,8)}$	$a_{(7,9)}$
$a_{(8,0)}$	$a_{(8,1)}$	$a_{(8,2)}$	$a_{(8,3)}$	$a_{(8,4)}$	$a_{(8,5)}$	$a_{(8,6)}$	$a_{(8,7)}$	$a_{(8,8)}$	$a_{(8,9)}$
$a_{(9,0)}$	$a_{(9,1)}$	$a_{(9,2)}$	$a_{(9,3)}$	$a_{(9,4)}$	$a_{(9,5)}$	$a_{(9,6)}$	$a_{(9,7)}$	$a_{(9,8)}$	$a_{(9,9)}$

Figure 4.10. Ideal Access Pattern After Transpose Operations for Weight Updates

The memory access pattern for a weights matrix W^2 with gradient matrix G scaled by the input layer A^1 is shown in Figure 4.10. It is equivalent to performing the following set of matrix operations from Equation (4.20) where in the gradient matrix G is equivalent to the scaled matrix $\eta\left(\hat{Y}-Y\right)^T$. We demonstrate this through the example 10 by 10 weights matrix, 10 by 10 gradient matrix and 10 by 10 input matrix. The forward propagation rule in Equation (4.11) does not require the matrices to be transposed as the input matrix A^1 is already in the correct format for the matrix operations for forward propagation throughout the neural network layers. On the other hand the backpropagation rule requires a series of dot products that must be computed with contiguous memory access patterns to maximise performance [84].

4.1.9 Solving the AOS to SOA Transformation Problem

In fact the problem of transposing matrices in the backpropagation rule to optimise memory access patterns to be contiguous can be generalised as a problem of transforming an Array of Structures (AOS) to a Structure of Arrays (SOA) [18]. We consider the fact that when performing forward propagation in the neural network, the input matrices H^1 and A^1 are in the AOS format. For forward propagation, the AOS format is necessary for calculating the accuracy of the model with respect to the ground truths. However, for backpropagation this presents a formatting problem as we transition to optimise for GPUs.

$$\begin{bmatrix} w_{(0,0)} & w_{(1,0)} \\ w_{(0,1)} & w_{(1,1)} \end{bmatrix} = \begin{bmatrix} w_{(0,0)} & w_{(1,0)} \\ w_{(0,1)} & w_{(1,1)} \end{bmatrix} - \eta \left(\begin{bmatrix} a^0_{(0,0)} & a^0_{(1,0)} \\ a^0_{(0,1)} & a^0_{(1,1)} \end{bmatrix} \cdot \begin{bmatrix} g^0_{(0,0)} & g^0_{(1,0)} \\ g^0_{(0,1)} & g^0_{(1,1)} \end{bmatrix} + \begin{bmatrix} a^1_{(0,0)} & a^1_{(1,0)} \\ a^1_{(0,1)} & a^1_{(1,1)} \end{bmatrix} \cdot \begin{bmatrix} g^1_{(0,0)} & g^1_{(1,0)} \\ g^1_{(0,1)} & g^1_{(1,1)} \end{bmatrix} \cdots \right.$$

Figure 4.11. 2x2 Weights Matrix Update in AOS Format

The main bottleneck in the backpropagation rule is the fact that if we want to update the weights in the neural network we must perform a series of dot products between the gradient matrix and the input matrix. However, as we increase the size of our training dataset, the number of shared memory updates when updating the weights matrices increases drastically, scaling with the number of individual threads in the GPU. This contention for shared memory updates can lead to a significant performance bottleneck if we overuse atomic update operations in the GPU [42]. The solution to this problem is to transform the AOS format of the input matrices and gradients to a SOA format and performing the dot product for each individual weight contiguously by transposing the matrices [84].

$$\begin{bmatrix} w_{(0,0)} \end{bmatrix} = \begin{bmatrix} w_{(0,0)} \end{bmatrix} - \eta \left(\begin{bmatrix} a^0_{(0,0)} & a^1_{(0,0)} & a^2_{(0,0)} & \cdots \end{bmatrix} \cdot \begin{bmatrix} g^0_{(0,0)} & g^1_{(0,0)} & g^2_{(0,0)} & \cdots \end{bmatrix} \right)$$

$$\begin{bmatrix} w_{(1,0)} \end{bmatrix} = \begin{bmatrix} w_{(1,0)} \end{bmatrix} - \eta \left(\begin{bmatrix} a^0_{(1,0)} & a^1_{(1,0)} & a^2_{(1,0)} & \cdots \end{bmatrix} \cdot \begin{bmatrix} g^0_{(1,0)} & g^1_{(1,0)} & g^2_{(1,0)} & \cdots \end{bmatrix} \right)$$

$$\begin{bmatrix} w_{(0,1)} \end{bmatrix} = \begin{bmatrix} w_{(0,1)} \end{bmatrix} - \eta \left(\begin{bmatrix} a^0_{(0,1)} & a^1_{(0,1)} & a^2_{(0,1)} & \cdots \end{bmatrix} \cdot \begin{bmatrix} g^0_{(0,1)} & g^1_{(0,1)} & g^2_{(0,1)} & \cdots \end{bmatrix} \right)$$

$$\begin{bmatrix} w_{(1,1)} \end{bmatrix} = \begin{bmatrix} w_{(1,1)} \end{bmatrix} - \eta \left(\begin{bmatrix} a^0_{(1,1)} & a^1_{(1,1)} & a^2_{(1,1)} & \cdots \end{bmatrix} \cdot \begin{bmatrix} g^0_{(1,1)} & g^1_{(1,1)} & g^2_{(1,1)} & \cdots \end{bmatrix} \right)$$

Figure 4.12. 2x2 Weights Matrix Update in SOA Format

The SOA format allows for each individual weight to be updated contiguously in memory. By doing so we can remove the contention for shared memory updates entirely which translates to a significant performance improvement in regards to the backpropagation rule. This is as a result that for our significantly large weight matrices, which are in order of 100×47 for the first layer and 47×47 for the second layer [29], performing the dot product of potentially millions of update nodes for each weight matrix can be extremely computationally expensive. Therefore, by utilising fast transpose operations with loop blocking techniques [27], we can significantly reduce the overhead of each weight and bias update, improving performance.

4.1.10 Neural Network Training Algorithm

We provide a generalised algorithm for training a neural network using the node aggregation function in Section (4.1.2), forward propagation rule in Equation (4.11), backpropagation rules defined in Section 4.1.6 and The MSE loss function in Equation (4.13). Given a graph based dataset G with node embeddings X and node labels Y. Furthermore, T and Y_T will be the test set and ground truth labels of the test set respectively with the assumption that both X and T are subsets of node embeddings of the entire graph G. We can then define the training algorithm for the neural network alongside its application on a given test set:

Algorithm 4 Neural Network Algorithm

function TrainTest(G, X, Y, T, Y_T, η, *epochs*)
  $W^1 \leftarrow$ Random Initialisation
  $W^2 \leftarrow$ Random Initialisation
  $b^1 \leftarrow$ [0.0f, 0.0f, ...]
  $b^2 \leftarrow$ [0.0f, 0.0f, ...]
  $H \leftarrow \text{GraphConvolution}(G, X)$
  for $i \leftarrow 1$ to *epochs* **do**
    $\hat{Y}, A^1 \leftarrow \text{ForwardPropagation}(H, W^1, W^2, b^1, b^2, A^1)$
    $L \leftarrow \text{MSE}(\hat{Y}, Y)$
    $A \leftarrow \text{Accuracy}(\hat{Y}, Y)$
    Print L, A
    $W^1, W^2, b^1, b^2 \leftarrow \text{BackPropagation}(H, A^1, \hat{Y}, Y, W^1, W^2, b^1, b^2)$
  end for
  $H_T \leftarrow \text{GraphConvolution}(G, T)$
  $\hat{Y}_T \leftarrow \text{ForwardPropagation}(H_T, W^1, W^2, b^1, b^2)$
  $L_T \leftarrow \text{MSE}(\hat{Y}_T, Y_T)$
  $A_T \leftarrow \text{Accuracy}(\hat{Y}_T, Y_T)$
  Print L_T, A_T
end function

Thus, concluding the background theory behind the neural network model, we have defined the forward propagation rule, node aggregation function, backpropagation rule and the training algorithm for the neural network. We can then define the implementation of the neural network model in the next chapter with the application of the model on a graph based dataset. Taking a working implementation, we can then evaluate its performance on the graph based dataset and compare the performance results of the model whilst developing out optimisation techniques using profiling tools to reduce overall training times.

4.2 Implementation

The structure in terms of development of the neural network training algorithm followed closely to the structure for the development of the graph convolution algorithm in Section (4.1) with code development in C++ and the Kokkos performance portable framework as to how the KNN algorithm was developed in Chapter 3. However, on top of the Kokkos optimised version for both CPUs and GPUs, we also develop a CUDA optimised version in tandem in order to utilise Tensor Core programming for faster matrix operations to guage the performance of the neural network model on newer Nvidia GPUs [51].

4.2.1 Dataset

We aim to apply the graph convolutional network model on the graph based dataset "Open Graph Benchmark Amazon Products" dataset [29]. The dataset is a graph based dataset where in the nodes within the graph represent products and the edges represent the co-purchase relationships between the products. That is where in if two products are connected by an edge, it means that the two products have been purchased together by a user within the same basket. The dataset contains 2,449,029 nodes or products and 61,859,140 edges or co-purchase relationships between the products. The dataset also contains node features for each product in the form of a 100-dimensional vector representing the product represented as 32-bit floating point values. Utilising the graph based dataset, we aim to predict the product category of each product in the dataset using the graph convolutional network model. The product categories are represented as a one-hot encoded vector where in each product can belong to one of 47 product categories [29].

Product Class	**Product Category**	**Product Class**	**Product Category**
0	Home & Kitchen	24	Used & Rental Textbooks
1	Health & Personal Care	25	Appliances
2	Beauty	26	Kitchen & Dining
3	Sports & Outdoors	27	Collectibles & Fine Art
4	Books	28	All Beauty
5	Patio, Lawn & Garden	29	Luxury Beauty
6	Toys & Games	30	Amazon Fashion
7	CDs & Vinyl	31	Computers
8	Cell Phones & Accessories	32	All Electronics
9	Grocery & Gourmet Food	33	Purchase Circles
10	Arts, Crafts & Sewing	34	MP3 Players & Accessories
11	Clothing, Shoes & Jewelry	35	Gift Cards
12	Electronics	36	Office & School Supplies
13	Movies & TV	37	Home Improvement
14	Software	38	Camera & Photo
15	Video Games	39	GPS & Navigation
16	Automotive	40	Digital Music
17	Pet Supplies	41	Car Electronics
18	Office Products	42	Baby
19	Industrial & Scientific	43	Kindle Store
20	Musical Instruments	44	Buy a Kindle
21	Tools & Home Improvement	45	Furniture & Decor
22	Magazine Subscriptions	46	Other
23	Baby Products		

Table 4.1: Product Categories for the Amazon Products Dataset

The rationale behind utilising the Amazon Products dataset is to demonstrate the application of using graph node information to predict node properties throughout the dataset, or in this case unseen products that have been yet to be classified. The assumption made about utilising the edges is centered around the idea that products that are co-purchased together usually hold similar properties or characteristics. Similar to the Lipschitzness assumption made by KNN [31], where in we assume that nodes that exist in N-Dimensional space that are close to each other in terms of Euclidean distance are also likely to have similar properties. Nodes that are connected by an edge in the graph or are close to each other in terms of graph shortest path distance are also likely to have similar properties or characteristics [95].

4.2.2 Static Compressed Row Storage (CRS) Graphs

The graph based dataset is represented as a static compressed row storage (CRS) graph where in the node edge information is stored using the "Kokkos::StaticCrsGraph" data structure [40]. Important to note is that within the CRS graph implemented in Kokkos, similar to how "Kokkos::Views" are implemented, they can be defined to store on a specific memory space such as Host CPU or CUDA Device GPU memory [40]. Efficient use of graph storage is important in the context of graph convolutional network training as the graph structure can massively impact the memory usage. If utilising a standard adjacency list representation for the graph, the memory usage would be of order $O(|V|^2)$ where in $(|V|)$ is the number of nodes in the graph [35].

The CRS graph representation is more memory efficient as it only stores the edges and node connections in the graph. The CRS graph representation is of order $O(|E|)$ where in $(|E|)$ is the number of edges in the graph [35]. The reduction in memory usage with the bound being of order $(|E|)$ is based upon the assumption that the vast majority of graphs in the context of real world datasets are sparse graphs where in the number of edges is significantly less than the number of nodes in the graph [95].

```
// For the graph structure
// To ensure that the graph is undirected, we add edges in both directions
std::vector<std::vector<int>> graph_edges(num_vertices * 2);

// Read edge information from the ''Products'' dataset
readGraphData(&graph_edges, ''edge.csv'', num_vertices, num_edges, ....);

// Using Kokkos to create a static CRS graph
using namespace Kokkos;

// Define Kokkos Graph Type
#define MemLayout LayoutRight
#define MemSpace CudaSpace
#define MemTrait MemoryTraits<RandomAccess>
using KokkosGraphType=StaticCrsGraph<int,MemLayout,MemSpace,MemTrait,int>;

// Create the CRS Graph
KokkosGraphType d_graph;
d_graph = create_staticcrsgraph<KokkosGraphType>("d_graph", graph_edges);
```

Listing 4.1: Constructing a Static CRS Graph in Kokkos

The CRS graph representation is implemented in Kokkos as a templated class where in the template parameters impact the memory layout, memory space and memory traits of the graph [19]. The main focus for GPU-graph optimisations is the utilisation of the "Kokkos::RandomAccess" which is used to specify strided or non-contiguous memory access patterns for the graph data structure.

```
for (int i = 0; i < array.length; i+=4) {
  // perform operation on array[i];
}
```

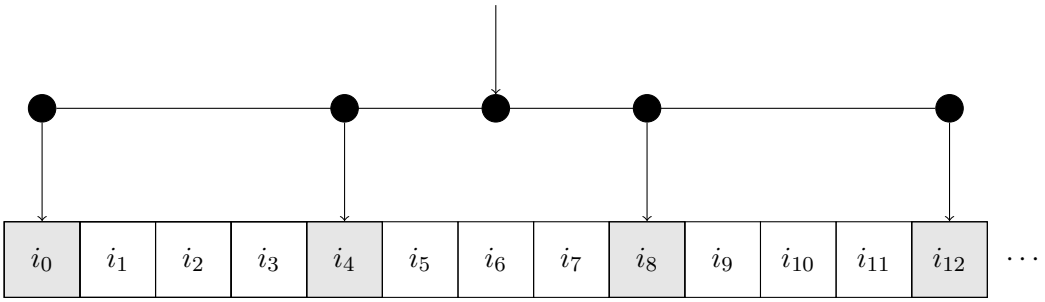


Figure 4.13. Example of non-contiguous (strided) memory access pattern for a 1D array

4.2.3 Node Aggregation

We implement the node aggregation function utilising C++ with Kokkos performance portable optimisations and use of the efficient "Kokkos::StaticCrsGraph" data structure for the graph representation [40]. The node aggregation function is effectively implemented as the sparse-matrix dense-matrix multiplication operation where in the sparse-matrix is the graph adjacency matrix and the dense-matrix is the node embeddings matrix. The node aggregation function only runs for the number of convolution operations specified by the user in the neural network training algorithm. We implement the normalisation trick introduced by Kipf and Welling where in the node embeddings are scaled by the inverse of the square root of the degree of the node in the graph in order to reduce the influence of nodes with high degrees in the graph and normalise the weightings towards each node neighbour [34].

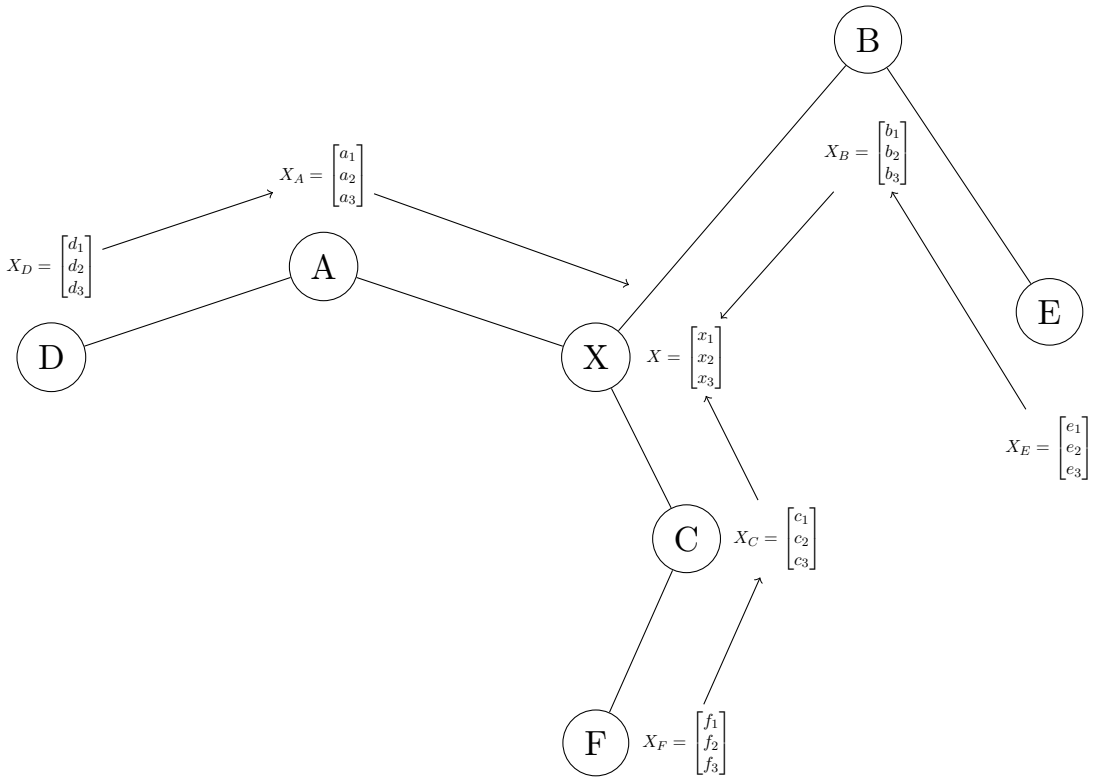


Figure 4.14. Example of Node Aggregation Function for a Node in a Graph

```
using Kokkos::parallel_for;
using Kokkos::parallel_reduce;
using Kokkos::TeamPolicy;
using Kokkos::TeamThreadRange;
using Kokkos::ThreadVectorRange;
using Kokkos::sqrt;

using ExecutionSpace = MemSpace::execution_space;

#if defined(KOKKOS_ENABLE_CUDA) && defined(KOKKOS_ENABLE_OPENMP)

parallel_for(TeamPolicy<ExecutionSpace>(node_embed, Kokkos::AUTO),
KOKKOS_LAMBDA (const TeamPolicy<ExecutionSpace>::member_type& team1) {
  int i = team1.league_rank();
  float normalised_length_1 = Kokkos::sqrt((float)(graph.rowConst(i).length
   + 1));
  parallel_for(TeamThreadRange(team1, feature_size), [=] (int j) {
    node_features(i, j) = node_features(i, j) / normalised_length_1;
  });
});

parallel_for(TeamPolicy<ExecutionSpace>(node_embed, Kokkos::AUTO),
  KOKKOS_LAMBDA (const TeamPolicy<ExecutionSpace>::member_type& team1) {
    int i = team1.league_rank();
    auto row_view_level_1 = graph.rowConst(i);
    int length_1 = row_view_level_1.length + 1;
    if (length_1 == 0) {
      // if the node has no neighbours, just use the node features
      parallel_for(TeamThreadRange(team1, feature_size),
        [=] (int j) {
          node_embeddings_layer_1(i, j) = node_features(i, j);
      } );
      return;
    }
    float length_1_div = (float)length_1;

    // for each of the neighbours:
    parallel_for(TeamThreadRange(team1, feature_size),
      [=] (int k) {
        float sum = 0.0f;
      parallel_reduce(ThreadVectorRange(team1, length_1 - 1),
        [=] (int j, float& temp_sum) {

          // non-contiguous memory access pattern
          temp_sum += node_features(row_view_level_1(j), k);
      }, sum);
      node_embeddings_layer_1(i, k) = sum;

      // also take into account the source node features
      node_embeddings_layer_1(i, k) += node_features(i, k);

      // all nodes have equal weights
      // Divide the node embedding by the number of nodes in the
   neighbourhood
      node_embeddings_layer_1(i, k) /= sqrt((float)length_1_div);
    });

    return;
});

#endif
```

Listing 4.2: Node Aggregation Function for Graph Convolutional Networks

Similar to the utilisation of hierarchical parallelism in the KNN algorithm in Chapter 3, we can also utilise hierarchical parallelism in the node aggregation function for the graph convolutional network with respect to modern GPU architectures. At the top level of the parallelism hierarchy, Kokkos launches teams of parallel threads in order to aggregate the node information for each node in the graph [45]. Beyond this, Kokkos then launches the parallel threads within each team to perform the sparse-matrix dense-matrix multiplication operation for each node neighbour in the graph [45].

In this context the parallelism hierarchy is vital in utilising maximal performance from the GPU architecture in the case where in a team of threads encounters a celebrity node with a high degree in the graph. This is due to the fact that if the parallelism hierarchy were to stop at the team level, the threads within the team would be idle as they would have to wait for the celebrity node to finish its computation before moving onto the next node in the graph. Instead within each nested team more parallel threads are launched to perform the sparse-matrix dense-matrix multiplication operation which negates the effect of celebrity nodes [39]. This is due to the fact that sufficient numbers of threads are launched with proportion to the size of the workload, thus reducing streaming multiprocessor (SM) idle time and increasing overall GPU utilisation [2].

4.2.4 Forward Propagation Kokkos-Optimised

The forward propagation rule implemented from Equation (4.11) is implemented in C++ as a kernel function utilising the Kokkos Kokkos Core libraries and nested parallelism for the graph convolutional network. The forward propagation rule is effectively implemented as a series of matrix operations followed by the application of a non-linear activation function to the output of the matrix operations.

```
parallel_for(TeamPolicy<ExecutionSpace>(node_embed, Kokkos::AUTO),
  KOKKOS_LAMBDA (const TeamPolicy<ExecutionSpace>::member_type& team1) {
    int i = team1.league_rank();
    // for each neuron in the hidden layer calculate neuron output
    parallel_for(TeamThreadRange(team1, neurons_per_layer),
      [=] (int j) {
        float sum = 0.0f;
        parallel_reduce(ThreadVectorRange(team1, feature_size),
          [=] (int k, float& temp_sum) {
            temp_sum += H(i, k) * W1(k, j);
        }, sum);
        A1(i, j) = sum + b1(j);
    });
});
```

Listing 4.3: Forward Propagation Through a Single Hidden Layer with Bias

We can break down the forward propagation rule at the ThreadVectorRange level where in each thread within a team is responsible for computing the dot product for the output of a single neuron in the hidden layer before applying the bias term. At the ThreadVectorRange level we only consider a single neuron in the hidden layer which can be visualised as a single column in the weights matrix W^1.

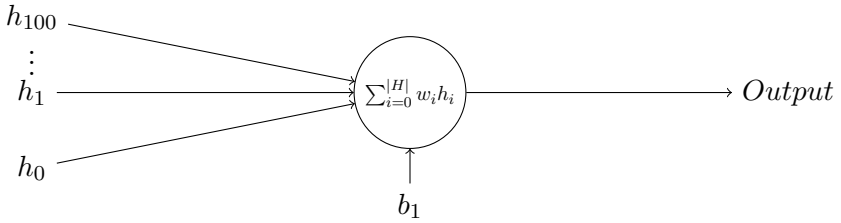


Figure 4.15. Single Neuron in the Hidden Layer at the ThreadVectorRange Level

4.2.5 Performance Portable Reduction Sums

One of the key optimisations used throughout the implementation of the graph convolutional network was "Kokkos::parallel_reduce" which is ubiquitous to the reduction sum operation which can be found in OpenMP [10]. Generally, reduction operations can be used to compute dot products between two vectors or to sum the elements of a vector. The "Kokkos::parallel_reduce" operation can in fact be used to perform a reduction using any associative binary operation. Furthermore the difference between the "Kokkos::parallel_reduce" and the OpenMP reduction operation is that the Kokkos version is more flexible in terms of the types of operations and execution space that can be used. It is also portable across different architectures and will be performant on both CPUs and GPUs [39].

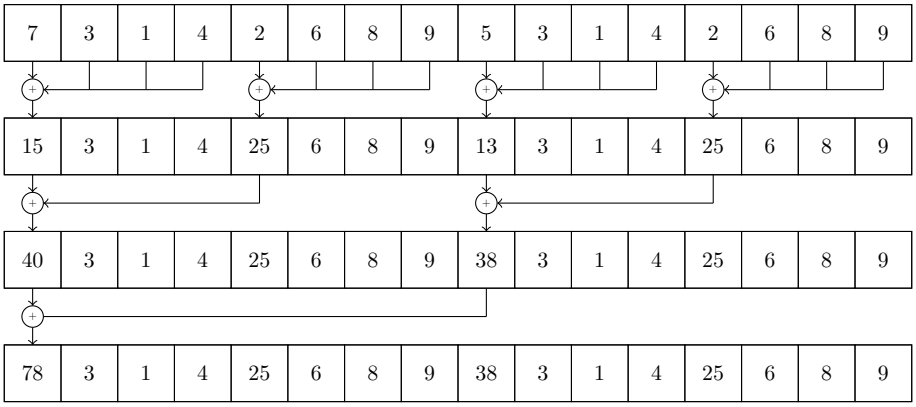


Figure 4.16. Cached Access Pattern for Reduction Operations

In fact, the Kokkos reduce operator is implemented differently dependent on the architecture being implemented for. This is as a result of the different memory hierarchies and memory access patterns that exist between CPUs and GPUs which can impact the performance of the reduction operation [43]. When considering the CPU memory architecture, we realise that the CPU has a cache hierarchy with multiple levels of cache which can be used to store intermediate results of the reduction operation [84]. However, CPUs are significantly better at performing contiguous memory access patterns as a result of this. Therefore, when utilising the reduction operation on a CPU host memory space, the reduction operation is implemented utilising a cache-friendly algorithm that is optimised for the CPU memory architecture [45].

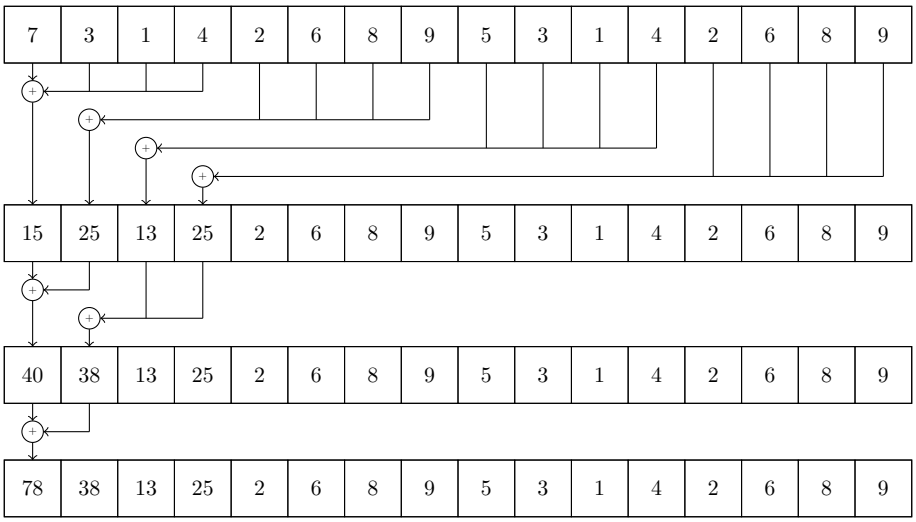


Figure 4.17. Coalesced Access Pattern for Reduction Operations

4.2.6 Memory Coalescing

Memory coalescing is a key optimisation technique that can be used to improve the performance of the reduction operation and makes use of the global memory hierarchy of GPU architectures. Memory coalescing is the proces of aggregating all memory writes from threads within a warp into a localised memory transaction. The main problem with the access pattern from Figure 4.16 is that after each thread in the warp reads from global memory, the threads in the warp will write to global memory in a non-contiguous manner. Thus when the next iteration of warps reads from global memory, the global memory access pattern will be non-coalesced which can reduce performance by up to 10× dependent on the architecture [79]. Figure 4.17 shows the coalesced memory access pattern where all threads in the warp read from global memory in a contiguous manner and write to global memory in a contiguous manner ensuring that the global memory access pattern is coalesced for every iteration of the reduction operation.

4.2.7 Backward Propagation Kokkos-Optimised

The kernel implemented for the backward propagation rule makes use of the update rules defined for W^1, b^1, W^2 and b^2. We implement the update rules by first taking advantage of converting our problem from an AOS to a SOA format as defined in Section 4.1.9. This is done by first computing gradients with respect to the output layers and the cross entropy loss function by merely subtracting the one hot encoded output from the predicted output [37]. We then perform the necessary transpose operations to guarantee fast contiguous access with the SOA format. The rest of the gradient calculations are done by implementing the matrix operations defined by Equations (4.19), (4.20), (4.21) and (4.22) utilising nested parallelism. Listing 4.4 demonstrates a simplified version of the kernel function implemented for the backward propagation rule for Equation (4.20).

4.2.8 Multi-Dimension Reduction Sum

```
// Kokkos::AUTO decides the thread launch configuration based on the number
    of threads
parallel_for(current_range_policy(total_updates_L2, Kokkos::AUTO),
  KOKKOS_LAMBDA (const current_range_policy::member_type& team1) {

    int rank = team1.league_rank();
    int j = rank % neurons_per_layer;
    int k = rank / neurons_per_layer;

    float sum_outer = 0.0f;

    // Perform the reduction sum for the outer loop
    parallel_reduce(TeamThreadRange(team1, num_vertices),
      [=] (int i, float& temp_sum_outer) {
          temp_sum_outer -= G_2_transpose(i, j) * A_1_transpose(i, k);
    }, sum_outer);

    // Ensures all teams have finished the reduction sum
    team1.team_barrier();

    // Only one thread update per team
    single(PerTeam(team1), [=] () {
      atomic_fetch_add(&weights2(k, j), sum_outer * learning_rate);
    });
});
```

Listing 4.4: Multi-Dimension Reduction Sum

4.2.9 Tensor Core-Optimised Graph Convolutional Networks

Tensor cores are specialised hardware units designed specifically for matrix operations and are present within Nvidia GPUs. Alongside the Kokkos-optimised implementation, we also developed a GCN implementation that utilises the tensor cores through the use of the cuBLAS library [56]. The library consists of a set of linear algebra routines which are imperative to the operation the Gradient Descent algorithm for the forward and back-propagation rules. However, we also consider that by developing a GCN implementation that utilises the tensor cores we sacrifice the performance portability that Kokkos provides [19].

On the other hand, we consider that the Kokkos-Kernels library provides a set of optimised linear algebra routines recently added support for utilisation of mixed precision tensor core operations [75]. For the purpose of this project, we will consider the cuBLAS implementation as a baseline for the performance of the Kokkos-optimised implementation. For forward and backpropagation rules, we make use of the cuBLAS GEMM-Ex operations which are generalised matrix-matrix multiplication operations followed by a bias addition operation. We can then specify the use of tensor cores by setting appropriate heuristic flags in the cuBLAS library [56].

```
cublasGemmEx(handle, CUBLAS_OP_T, CUBLAS_OP_N,
             num_vertices, hidden_layer_size, feature_size,
             &alpha,
             x, CUDA_R_32F, feature_size,
             weights_1, CUDA_R_32F, feature_size,
             &beta,
             biases_1, CUDA_R_32F, n,
             CUDA_R_32F, CUBLAS_GEMM_DEFAULT_TENSOR_OP);
```

Listing 4.5: Tensor Core-Optimised Forward Propagation Rule

4.2.10 CUDA Deep Neural Network Library

For implementation of activation functions, we make use of the CUDA Deep Neural Network Library (cuDNN) alongside some routines programmed directly in CUDA [61]. The cuDNN library is a GPU-accelerated library that provides a set of routines for computing the output of our activation functions. It is introduced as an alternative to programming the calculations directly such as in our Kokkos optimised version. As an example we can make use of the cuDNN activation forward function to compute the output of the Softmax activation function output with input normalisation after a hidden layer matrix multiply routine.

```
cudnnSoftmaxForward(cudnn, algo, CUDNN_SOFTMAX_MODE_CHANNEL,
                    &alpha_activation, input_desc, softmax_input,
                    &beta_activation, output_desc, softmax_output);
```

Listing 4.6: Softmax Activation Function [58]

A performance portable version of the tensor core-optimised implementation can also be feasibly implemented using the Kokkos-Kernels library. In fact an extension to Kokkos that is meant to operate specifically on tensor formatted data has been developed recently funded by the US Department of Energy [80]. The KokkosTensor extension would give access to tensor operations that are optimised for the tensor cores present in Nvidia GPUs similar to how one can program directly to tensor cores using WMMA (Warp Matrix Multiply Accumulate) operations in CUDA [57]. The KokkosTensor extension would provide a performance portable implementation of GCN training that would be able to run on CUDA, HIP and OpenMP backends [19].

4.2.11 Runtime Parameters

The testing of the GCN implementation was done similarly to how we tested the KNN algorithm in Chapter 3. We considered the effect of varying the number of training nodes alongside the number of test nodes to act as a validation set. Again, we are utilising the University of Warwick's DCS GPU cluster for the testing of the GCN implementation in a single node configuration [70]. We consider test and train sets of sizes from 800 train 200 test nodes to 1638400 train nodes and 409600 test nodes respectively increasing in powers of 2 each time. As the size of the feature vectors for each node is 100 and there are 47 classes in the Open Graph Benchmark Products dataset [29], we consider that the neural network architecture will be similar to the implementation in Figure 4.8. This is where in the number of nodes after the first input layer is equivalent to the size of the number of feature vectors for each node. This is then followed by an output layer of size 47 which is identical to the number of classes in the dataset for the Softmax activation function for one-hot encoded vectors.

- Number of training nodes: Variant
- Number of test nodes: Variant
- Number of features: 100
- Number of classes: 47
- Learning rate: $\frac{1}{\text{Train Size}}$
- Number of epochs: 100
- Batch size: Full Batch
- Convolution Operations: 3

The runtime parameters for the Kokkos-optimised GCN implementation are given below in table 4.2. There are a few key reasons as to why we chose the parameters for test and train sizes as we did. The first reason for having a scaling factor of 2 for the test and train sizes is to ensure that we can see the effect of the scaling factor on the performance of the GCN implementation. The second reason alludes to the fact that we split our train and test sizes in ratios of 80% train and 20% test. This is mainly utilised for a validation set to ensure that the GCN implementation is generalising well to unseen data [78]. As an additional note for our runtime parameters some compiler flags were added for the Tensor Core-optimised implementation to ensure that the compiler would be able to make use of the cuBLAS and cuDNN libraries for compilation [56][58]. These were: -lcudart -lcublas -lcuda -lcudnn for the Tensor-optimised implementation.

Version	Kokkos Optimised GCN
Partition	"Gecko"
CPU	AMD EPYC 7443
Core Count	24 Physical Cores, 48 Logical Threads
Total Host Memory	128GB
GPU	Nvidia A10 (24GB)
Compiler	g++ GCC 12.2
CUDA Toolkit	CUDA Devkit 12.3
Kokkos Version	4.1.00
Compiler Flags	-O3 -std=c++17 -funroll-loops -fopenmp-simd -march=core-avx2 -mtune=core-avx2 -mrtm -fopenmp
Test Repeats	Average of 5 Runs

Table 4.2: Runtime Parameters for Kokkos Optimised GCN Algorithm

4.3 Neural Network Train Time Results

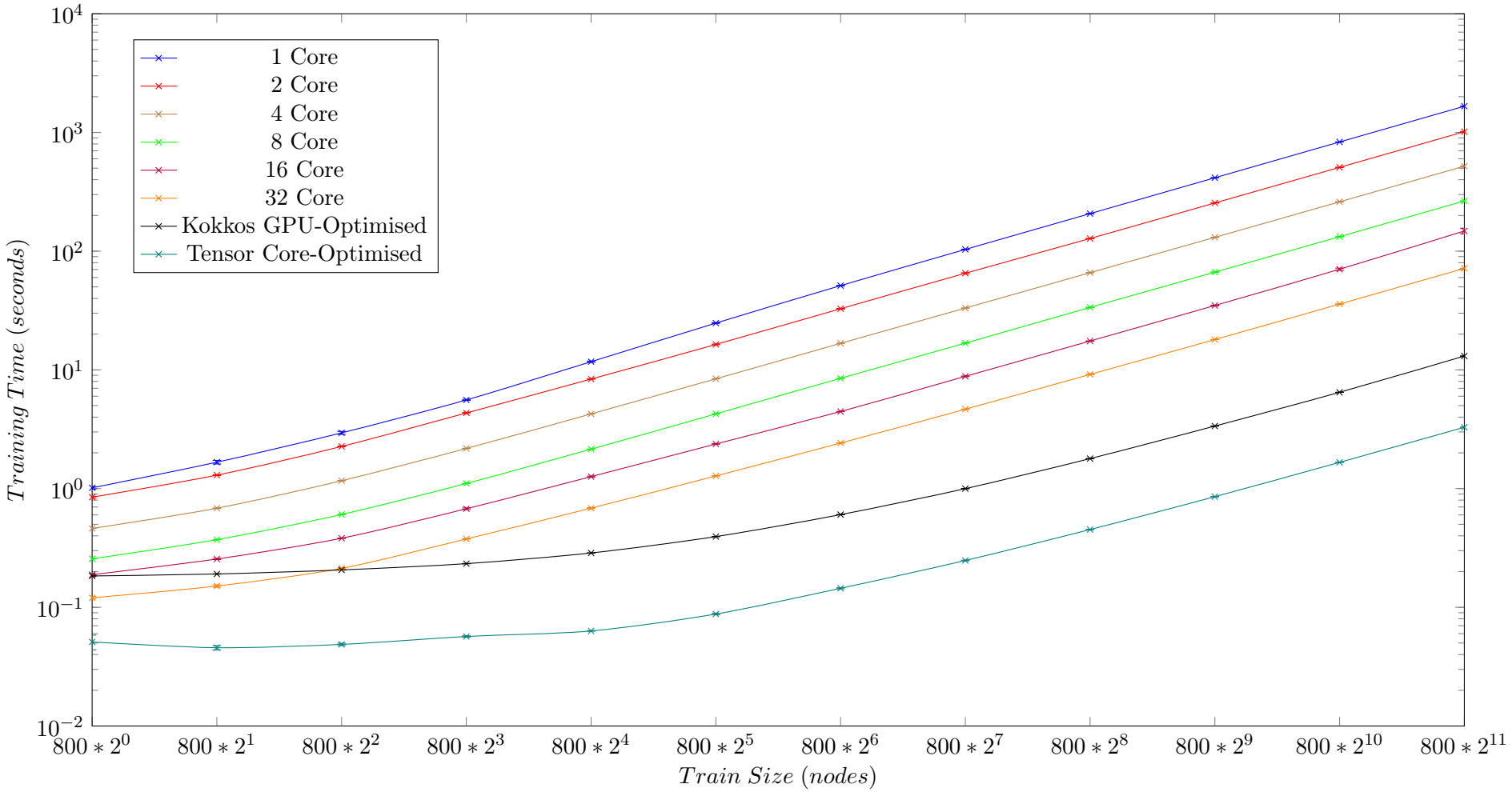


Figure 4.18. Kokkos-Optimised vs CUDA Tensor Core-Optimised

4.3.1 Forward Propagation Timings

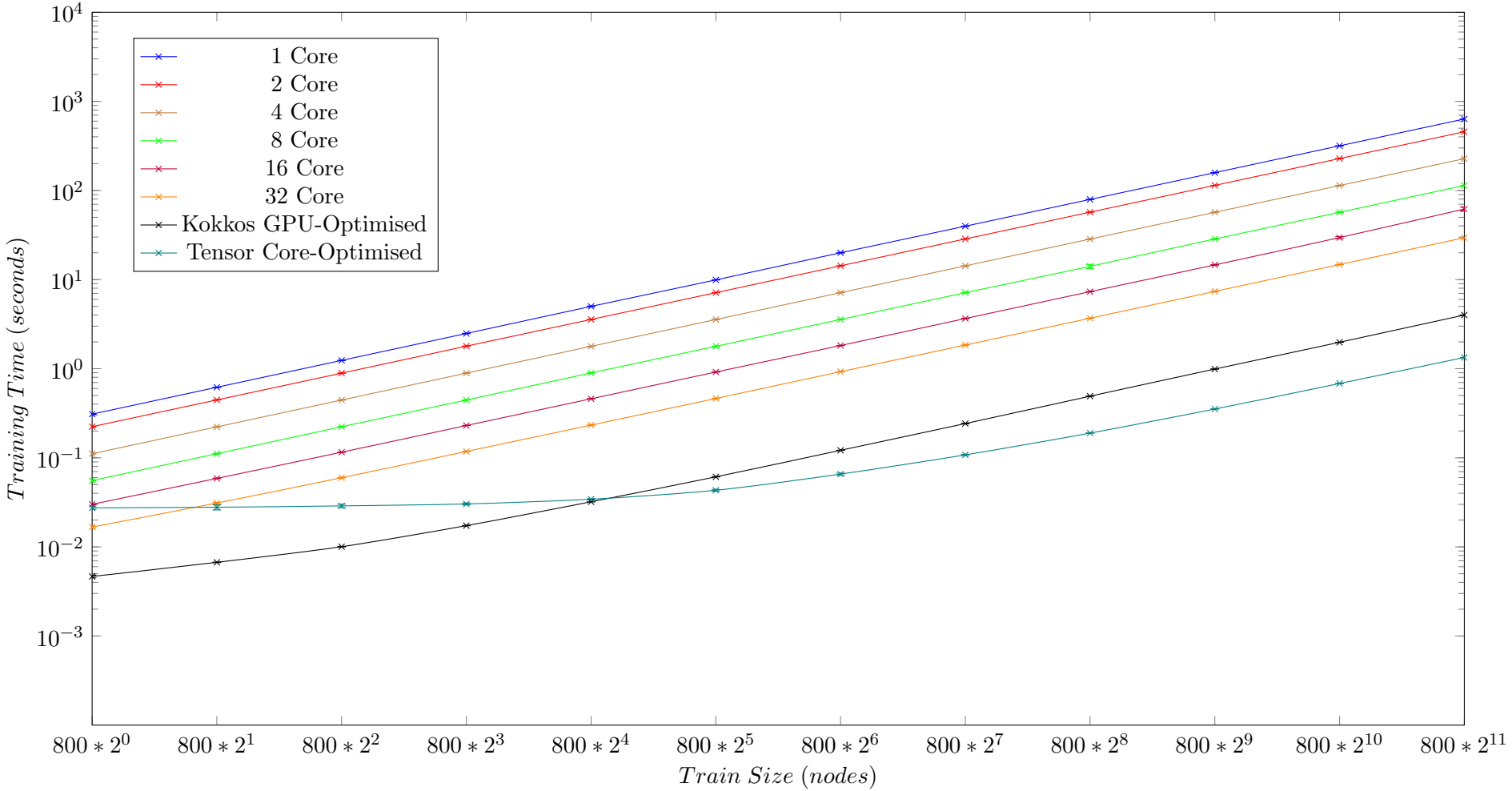


Figure 4.19. Forward Propagation Timings

4.3.2 Backward Propagation Timings

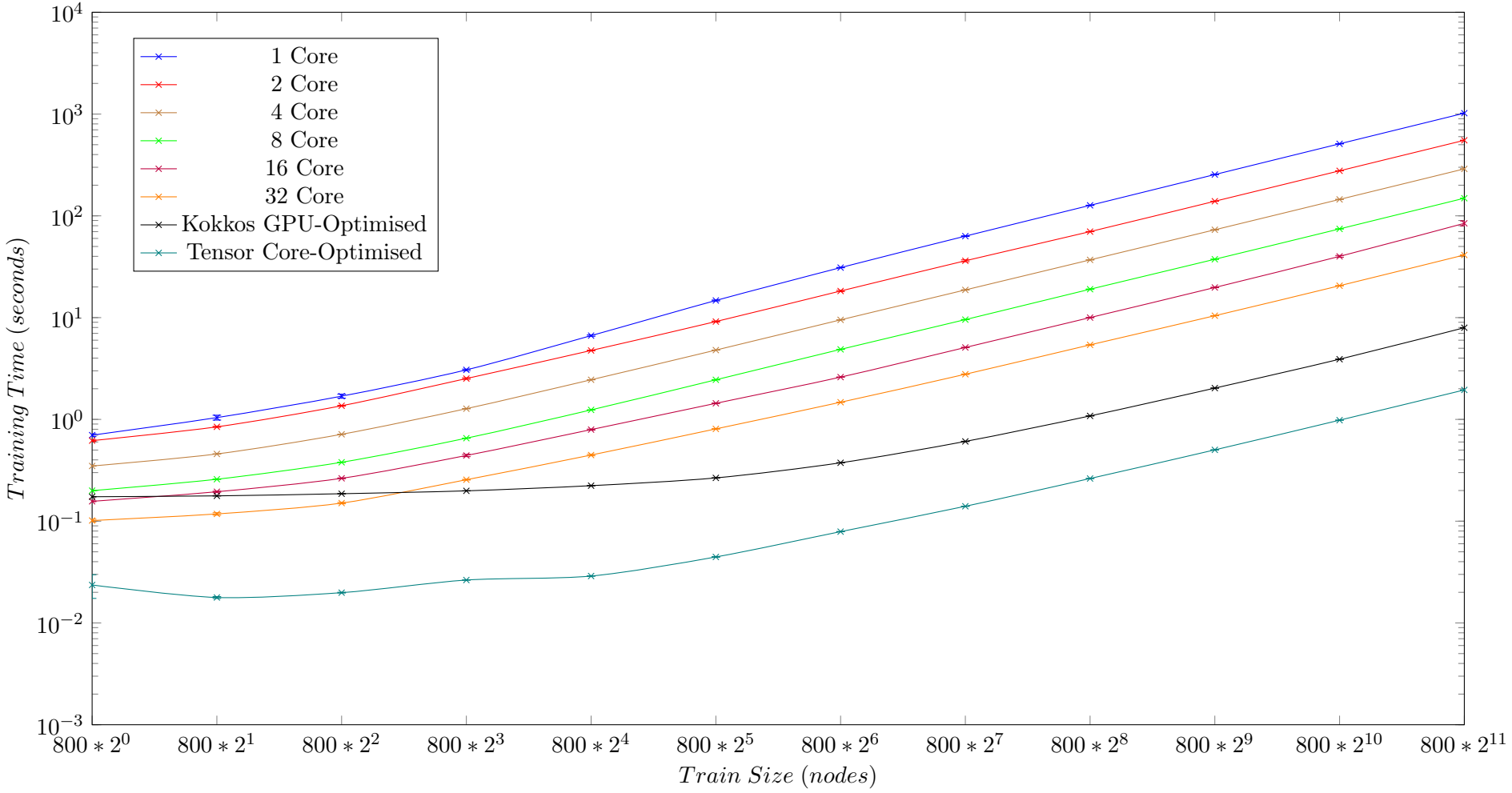


Figure 4.20. Backward Propagation Timings

4.3.3 Neural Network Training Analysis

We find in general the scaling of the speedup in proportion to the problem size with regards to the number of cores demonstrates a linear speedup where in the best case scenario the speedup is 32× when utilising 32 cores over 1 core. This is also known as strong scaling where the problem size is fixed and the number of cores is increased. Strong scaling is prolific in regards to the area of measuring performance in parallel computing [27].

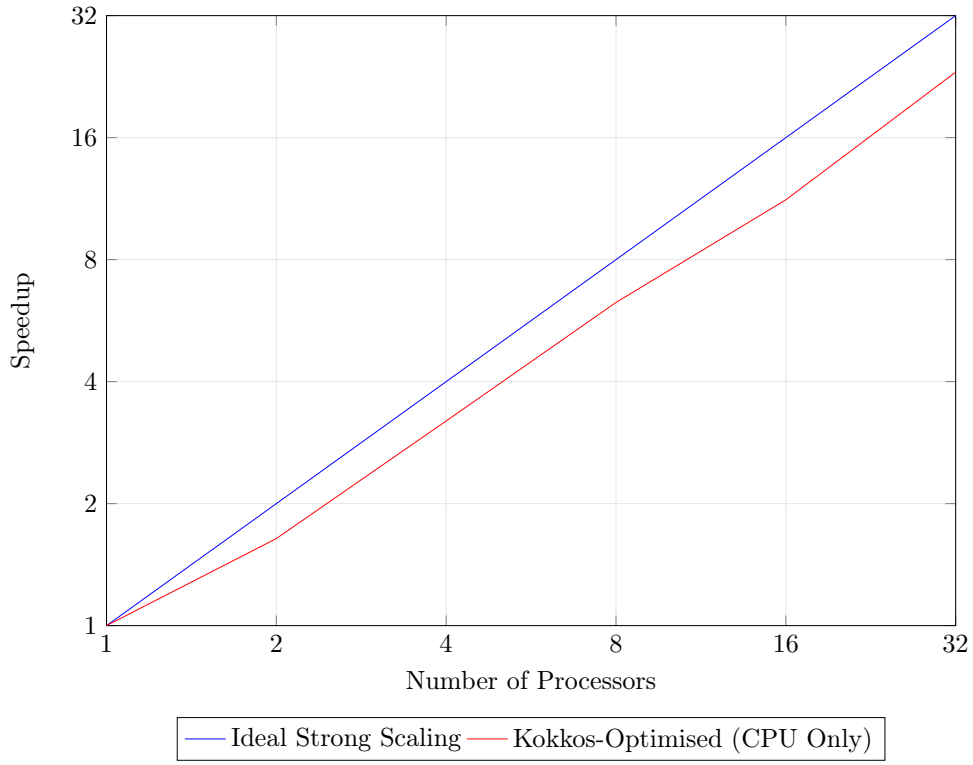


Figure 4.21. Strong Scaling Graph on Largest Problem Size

The reasoning as to why the speedup scales so well in regards to the number of cores is due to the fact that the neural network training operation itself is highly parallelisable. The neural network training operation is composed of many matrix multiplications and vector operations which can be parallelised across multiple cores. Furthermore, as the algorithm is iterative in nature and does not suffer from any data dependencies between iterations, in regards to Amdahl's Law [84], the parallelisable fraction of the algorithm is high.

$$S(p) = \frac{1}{(1 - P) + \frac{P}{p}} \tag{4.24}$$

This is where in the speedup equation $S(p)$, P is the fraction of the algorithm that can be parallelised and p is the number of processors.

On the other hand, when we consider the smaller problems running on the Kokkos-optimised implementation on the GPU, we find that the speedup is not as optimal as a result of overheads which are explained in the backward propagation analysis. However, as we increase the problem size beyond this point, we find that the timings linearly increase with problem size on all configurations. This is due to the fact that the problem size is now large enough to offset of the overhead brought upon the utilisation of our Kokkos-optimised backward propagation implementation.

4.3.4 Total FLOPs Calculation

We first take the number of calculations required for forward propagation which can be calculated by the number of calculations required for each layer. We make the assumption similar to the one made in the KNN algorithm that we treat multiplies and adds as separate operations as we assume that we do not use fused multiply-add operations. Weights W^1 and W^2 have 47×100 and 47×47 neurons at their respective layers each with bias vectors of size 47. We choose to ignore the the FLOP count for Leaky ReLU as it is negligable in comparison to the matrix multiplications and vector operations. Additionally, it is impossible to determine the number of FLOPs required for the Leaky ReLU operation as it is dependent on the number of negative values in the matrix. On the other hand, our Softmax operation requires 46 additions, 47 exponentials, 47 divisions alongside 47 subtractions for input normalisation. In the case of the largest problem size of 1,638,400 nodes we have the following:

$$\text{FLOPs} = 2 \times 1,638,400 \times (4,700 + 2,209) + 1,638,400 \times 281 = 23.1 \text{ GFLOPs} \quad (4.25)$$

We then take the number of calculations required for backward propagation which can be calculated by the number of calculations required for each layer. For brevity given the similarity in the number of calculations required for each instance of forward and backward propagation, although we do have to account the additional calculations required for gradients, which involves the multiplications of the gradients with the weight matrix W^2 of size 47×47 to propagate the gradients backwards to update W^1. In the case of the largest problem size of 1,638,400 nodes we have the following:

$$\text{FLOPs} = 2 \times 1,638,400 \times (4,700 + 2,209 + 2,209) + 1,638,400 \times 141 = 30.1 \text{ GFLOPs} \quad (4.26)$$

In turn, the total number of FLOPs performed for the largest problem size for 100 total epochs, therefore giving us 5.32 TFLOPs for the training phase, not inclusive of the graph convolution preprocessing step. We provide the TFLOP/s performance for the largest problem size for each configuration.

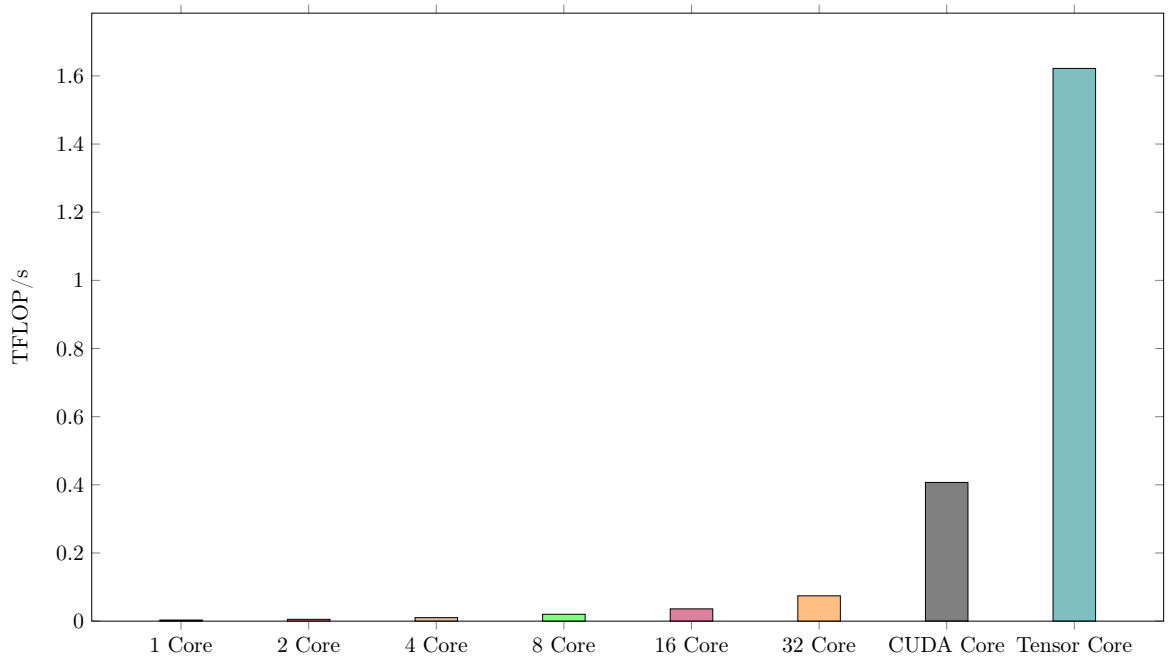


Figure 4.22. TFLOP/s Performance

4.3.5 Breaking the Teraflop Barrier

We find that the Tensor Core-optimised implementation is able to break the Teraflop barrier with a performance of 1.62 TFLOP/s given the largest tested problem size. In fact, the performance is 4× greater than the Kokkos-optimised implementation running on the Nvidia A10. This is due to two main following reasons:

- The first being that the library-optimised cuBLAS functions such as cublasGemmEx are highly optimised specifically for the Nvidia A10 GPU with certain algorithmic optimisations such as the utilisation of Strassen's algorithm for matrix multiplication [56].
- The second being that Tensor Cores in general are expected to provide a 4× to 5× speedup over traditional CUDA cores when applied for matrix multiplication operations [51].

In comparison, the Kokkos-optimised implementation running on the Nvidia A10 GPU is able to achieve a performance of 0.41 TFLOP/s. This difference in performance is non-trivial and has significant connotations for the field of artificial intelligence and machine learning. Especially for algorithms reliant on matrix operations such as deep neural networks. However, we have to consider the following tradeoffs when comparing both implementations:

- The Kokkos-optimised implementation is performance portable and can run on a variety of different architectures with minimal to no changes to the codebase. This is due to the fact that Kokkos is able to abstract the underlying hardware architecture from the developer [19].
- Whilst Tensor Cores are highly optimised for matrix multiplication operations, they are not as flexible as general purpose CUDA cores and are only useful for a subset of operations such as matrix multiplication [51].
- To further the previous point, not all artificial intelligence or machine learning operations are reliant on matrix operations in which case Tensor Cores would pose no benefit over traditional CUDA cores.
- The Tensor Core-optimised implementation may become obsolete in the future as newer architectures may not support Tensor Cores as the newest developments in the field of artificial intelligence and machine learning may not be reliant on matrix operations as the field is still rapidly developing.

To conclude this section, we find that the Tensor Core-optimised implementation is capable of breaking the Teraflop barrier on a single card, however, it sacrifices performance portability and application generalisation for performance. In contrast, the Kokkos-optimised implementation is able to provide adequate performance on a variety of different architectures in the HPC space. Combined with powerful abstractions such as the utilisation of hierarchichal parallelism and memory spaces, Kokkos is able to provide a powerful tool for the development of high performance scientific computing applications.

4.3.6 Graph Convolution Analysis

The results for the graph convolution operation are kept separate from the results for neural network training due to the fact that the graph convolution operation only has to be performed once for the entire training set before training the neural network. In this case, we perform the graph convolution operation for the entire Open Graph Benchmark Products dataset [29] using all 2,449,029 nodes and 61,859,140 edges. In both the Kokkos-optimised and Tensor Core-optimised implementations, we utilise the graph convolution operation from Listing 4.2. We find that for the when running multi-threaded on the Kokkos-optimised implementation that utilising the "Kokkos::Dynamic" scheduling algorithm provided the best performance over "Kokkos::Static". This is due to the fact that the work performed at each node is not uniform and the "Kokkos::Dynamic" scheduling algorithm works based on a workstealing queue which is more suited to the non-uniform workloads of the graph convolution operation [39]. When running the graph convolution operation on GPUs we find that the Nvidia A10 GPU can achieve a speedup of 8× over the Kokkos-optimised code compiled for OpenMP when utilising 32 cores.

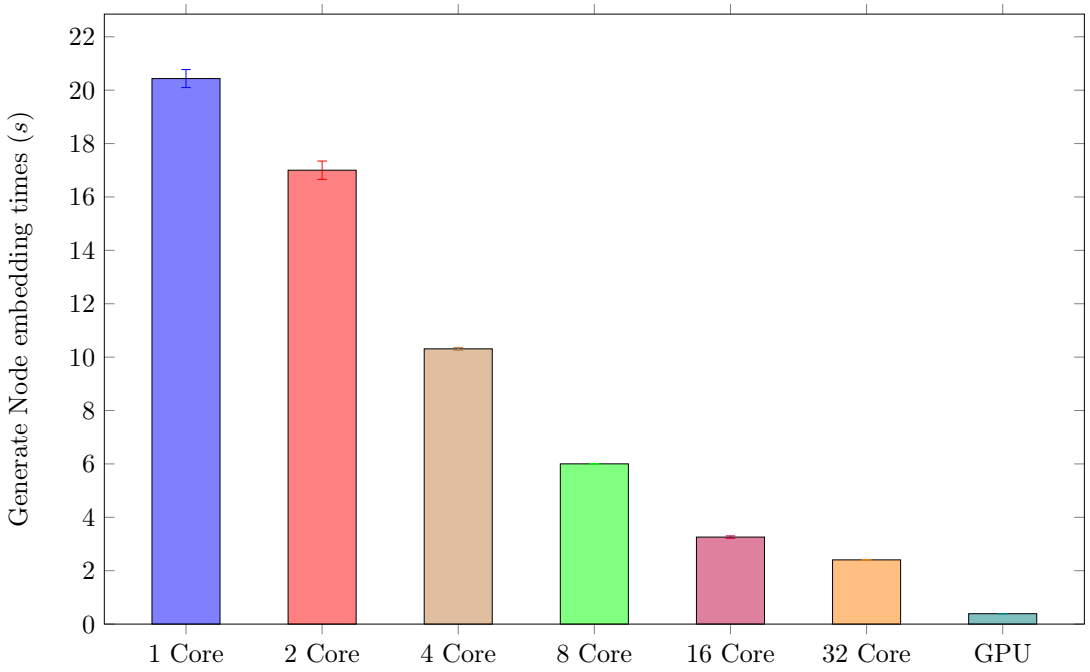


Figure 4.23. Generate Node Embedding Times

The results for the graph convolution operation after being ran 3 times on each configuration demonstrate that the application is more bound by memory bandwidth than compute. This is due to the fact that making non-contiguous memory accesses to host memory can cause a significant performance hit [84]. Additionally, we ensure that the tests were ran one after the other on the same node to ensure that no instance of thrashing occured from test to test. Previously, for the smaller core count test, we ran multiple tests at the same time on the same node which was less than ideal as this would cause thrashing to occur. We include these old results in the appendix for comparison. Finally, we also made a transition from the simple node average scheme to the normalisation scheme introduced in Section 4.1.1 [34]. The performance of the normalisation scheme to the simple node average scheme was negligable although the normalisation scheme was more computationally expensive due to the presence of an inverse square root. However, as stated previously the graph convolution operation is more bound by memory bandwidth than compute which negated the difference.

4.3.7 Additional Notes on Forward & Backward Propagation

Via observation of the forward and backward propagation timings, it was found that backward propagation was significantly slower as opposed to forward propagation as proven by the fact that the number of FLOPs required for backward propagation was greater than that of forward propagation. As an additional note, We observe that the Kokkos-optimised implementation running on smaller problem sizes on the GPU suffered from significant overheads which were not present on either the Kokkos-optimised implementation running on the CPU or the Tensor Core-optimised implementation. This was due to the fact that our application design specifically for the GPU with Kokkos utilised a decomposition approach that broke down the backpragation dot products into smaller dot products to be computed in parallel before being atomically added to maintain the integrity of the gradients.

4.3.8 Team Decomposition Trick

Effectively, one of the main bottlenecks in utilising "Kokkos::TeamPolicy" execution was that Kokkos specifically sets a maximum upper bound of 1024 threads per team [39]. This was a significant bottleneck as the number of threads was not sufficient to fully utilise the GPU's resources. In turn, taking the Multi-Dimension Reduction Sum from section 4.2.8 we utilise a trick which forces Kokkos to launch more teams to decompose the dot products. We effectively artifically launch more thread teams than the number of dot products that need to be computed based on the number of Streaming Multiprocessors (SMs) on the GPU [57]. This effectively allows us to fully utilise the GPU's resources and achieve a speedup of 3× over simply using Kokkos' default execution policy for large problem sizes.

However, to maintain concurrency between threads we utilised "Kokkos::atomic_add" operations to ensure that the gradients were not corrupted by race conditions which was a significant bottleneck in the implementation [42]. Finally, once this was combined with the overhead of launching this many thread teams for each iteration of backpropagation, on smaller problem sizes the overheads were significant enough to be noticeable at ~ 0.17 seconds for the smallest problem size of 800 nodes and above. However, for larger problem sizes the benefit of speedup from fully utilising the GPU outweighed the overheads. This justified the design decision to sacrifice performance on smaller problem to allow for performance scaling on larger problem sizes.

```
long long total_updates = hidden_layer_size * neurons_per_layer * num_SM;
parallel_for(current_range_policy(total_updates, Kokkos::AUTO),
  KOKKOS_LAMBDA (const current_range_policy::member_type& team1) {
    // Calculate start indexes and determine which weight matrix to update
    int start = ....
    float sum_outer = 0.0f;

    // Decompose the reduction operations into smaller reductions
    parallel_reduce(TeamThreadRange(team1, start, loop_size),
      [=] (int i, float& temp_sum_outer) {
        // perform the dot product for this partition
    }, sum_outer);

    // Ensures all thread teams have finished, maintain concurrency
    team1.team_barrier();
    single(PerTeam(team1), [=] () {
      atomic_fetch_add(&weights2(k, j), sum_outer * learning_rate);
    });
});
```

Listing 4.7: Team Decomposition Trick

4.4 Validation

In order to test and validate the correctness of the neural network model we have implemented, we utilise the fact that the Open Graph Benchmark Products dataset [29] has a set of training, validation and test datasets. These sets appear as contiguous blocks in the dataset and are labelled from nodes 0 to 196615 for the training set, 196616 to 235938 for the validation set and nodes 235939 to 2449029 for the test set. We tune our hyperparameters such as learning rate, number of epochs and the alpha value for the Leaky ReLU function to achieve the best validation and test accuracy possible available with our current model. However, we must also take into account that there exists slight variances in implementation between our implementation of Full Batch Gradient Descent on GCNs and the original implementation by Kipf and Welling [34].

Hyperparameter	**Value**
Learning Rate	$\frac{0.75}{\lvert\text{Train}\rvert}$
Number of Epochs	10,000
Leaky-ReLU Alpha Value	0.3
Bias	Disabled
Neighbourhood Sampling	Depth 5

Table 4.3: Hyperparameters for Training

The version of the GCN model we have implemented utilises a different data type for the weights and biases of the neural network as well as the fact that we have implemented the Leaky ReLU activation function as opposed to the ReLU activation function. Our version implemented in C++ makes use of single precision 32-bit floating point numbers as opposed to double precision 64-bit floating point numbers from the original implementation in Python [34]. This was done to reduce overall memory usage and increase performance of the application. This is in regards to the fact that GPUs can effectively double their throughput when using single precision floating point numbers as opposed to double precision floats [84]. The accuracy of the datatype can have a significant impact on the overall classification accuracy of the model as higher precision datatypes can lead to overfitting however, it can also lead to more accurate results in certain scenarios [28].

Dataset	**Full Batch GCN** [34]	**Our Tensor-Optimised Version**
Products Validation Set	92.00%	90.11%
Products Test Set	75.64%	75.97%

Table 4.4: Validation and Test Set Accuracy

In the end, we find that our model when trained with these hyperparameters our model is able to achieve similar if not better results than the original implementation by Kipf and Welling [34]. We achieve a slightly higher test set accuracy of 75.97% as opposed to the original implementation which achieved a test set accuracy of 75.64% [29]. This is a significant result as it demonstrates that our implementation is slightly more optimal at generalising to unseen data than the original. However, we must also take into account that our implementation has a slightly lower validation set accuracy of 90.11% as opposed to the original implementation which may be as a result of the slight variances in implementation between the two models.

5 Project Management

The project was managed following an incremental development approach with the project being broken down into smaller tasks which were then completed in a series of iterations. The project was managed using the Agile methodology with the project also taking elements of certain software development methodologies such as N-version programming [36], where in we develop multiple versions of the same software in order to compare the results and coalesce the best parts of each version to develop the final Kokkos-optimised version of the KNN & GCN models. The project was managed using the following tools:

- **Outlook** - Used for communication with the project supervisor and to schedule meetings [53].
- **Git** - Used for distribution of software and managing version control [12].
- **Texworks** - Used for the writing of the report [24].
- **Tabula** - Keeping track of supervisor meetings and record keeping of discussions [67].
- **Python** - Used to develop automated testing scripts for the KNN & GCN models to save time when outputting results for the report [89].

5.1 Project Plan & Methodology

Originally, the project was planned to be a completed series of iterations where in the first iteration would be a single threaded implementation of the selected machine learning algorithm. The second iteration would involved using OpenMP to parallelise the code to accelerate the performance of the application [10]. Finally, we would make use of the performance portable library Kokkos in order to develop a version of the application that could be able to run on a variety of different architectures such as GPUs and CPUs [19]. As an extension to this project, if time permitted, the development of an MPI version of the application which could run amongst a cluster of compute nodes would be developed for both multi-nodal GPU and multi-nodal CPU configurations [20]. However, due to time constraints encountered during the development of both the Kokkos-optimised KNN and GCN models, the MPI version of the application was not developed.

Although the project was originally meant to be completed iteratively for each version of the application, the project plan was also flexible enough as to allow changes to be made to older applications in order to improve the performance of said application if further optimisations could be made. This was the case for the KNN model optimised using Kokkos. Effectively, once a new optimisation technique was learned when developing the GCN model later on in the project, the same optimisation technique could be applied to the KNN model. As an example of this, the nested parallelism optimisation technique was learned when developing the GCN model was applied to the KNN model to achieve significant performance gains on certain configurations. Originally, the project was also planned with the use of a Gantt chart to keep track of the progress of the project, however, as development progressed, the Gantt chart ordering was altered multiple times throughout the project duration to reflect the changes in the project plan. The finalised Gantt chart changed significantly from the original plan as significant progress was made developing the Kokkos-optimised versions of the code.

5.2 Finalised Project Gantt Chart

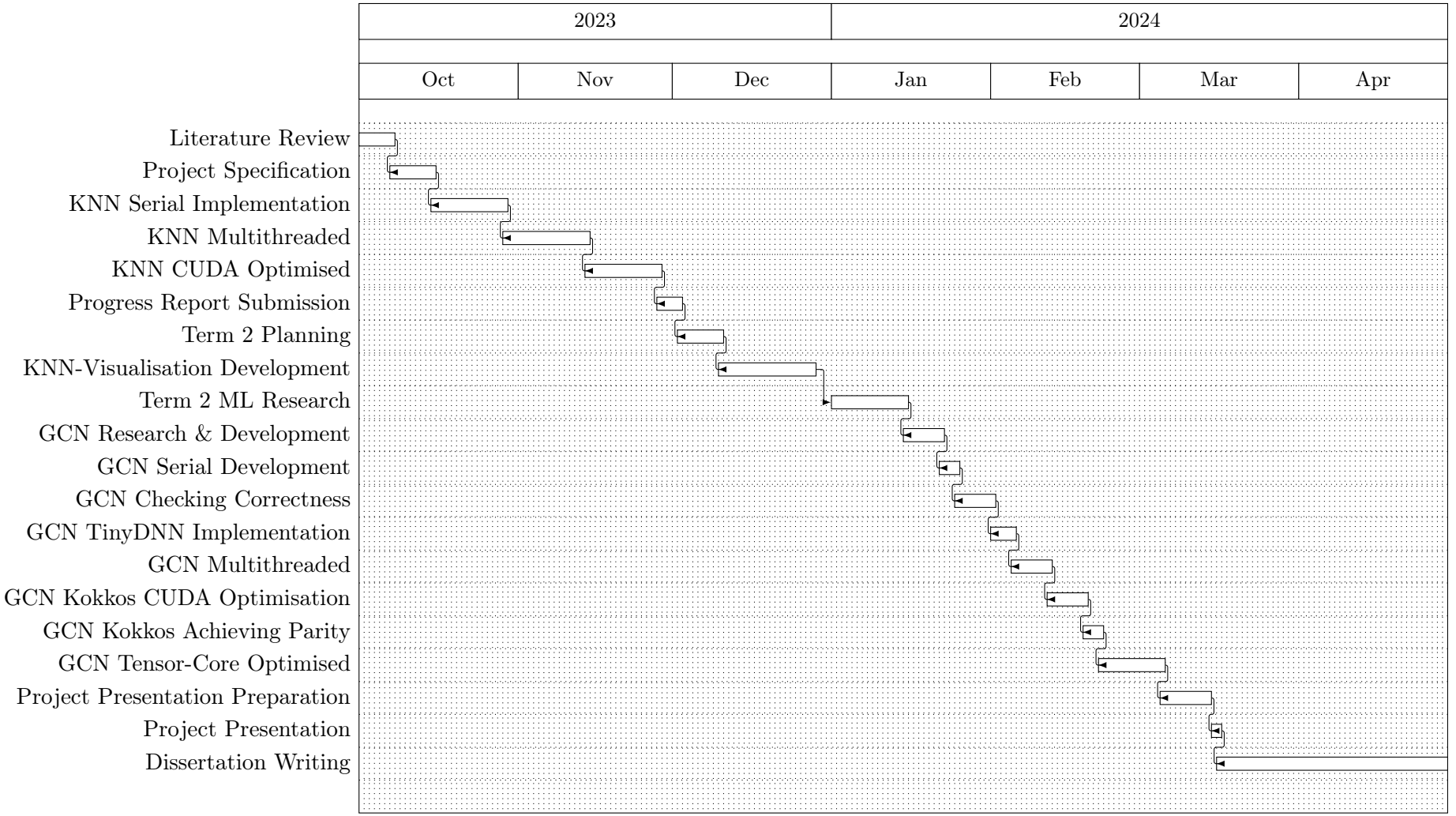


Figure 5.1. Finalised Project Gantt Chart

5.3 Project Tracking

We make use of the fact that due to the weekly meetings with the project supervisor, we were able to keep track of the progress of the project for each bimonthly period in term 1. Additionally, we transition to a weekly scheduled meeting with the project supervisor in term 2 to ensure that the project was on track for completion. Effectively, records of what was discussed during each meeting were kept using the Tabula application [67].

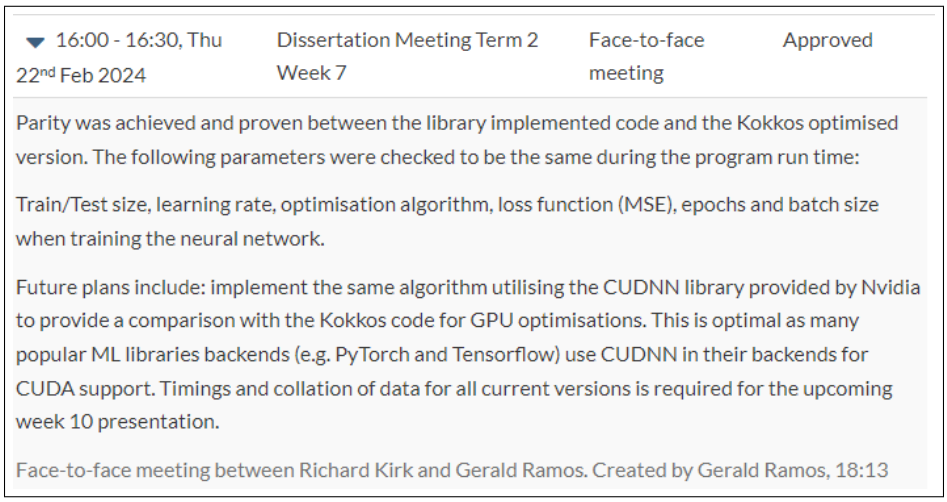


Figure 5.2. Tabula Meeting Record

5.4 Legal Issues

With regards to the legal issues surrounding this specific project, the main issue that arises was the fact that the Kokkos Core libraries are licensed under the BSD 3-Clause License [85]. This license allows for the software to be used for personal development and research so long as the original copyright notice is retained at the top of the source code for each version of the application one develops. All versions of each developed applications has the Kokkos Core license retained at the top of the main file of the application.

```
1  //
     ************************************************************************
2  //@HEADER
3  //
     ************************************************************************
4  //
5  //                        Kokkos v. 4.0
6  //       Copyright (2022) National Technology  Engineering
7  //               Solutions of Sandia, LLC (NTESS).
8  //
9  // Under the terms of Contract DE-NA0003525 with NTESS,
10 // the U.S. Government retains certain rights in this software.
11 //
12 // Part of Kokkos, under the Apache License v2.0 with LLVM Exceptions.
13 // See https://kokkos.org/LICENSE for license information.
14 // SPDX-License-Identifier: Apache-2.0 WITH LLVM-exception
15 //
16 //@HEADER
17 //
     ************************************************************************
```

Listing 5.1: Kokkos Core License

This listing demonstrates the Kokkos Core license present in all versions of the developed applications inclusive of the KNN and GCN models. Legally, there are no other issues involved with the development of the project as Kokkos was the only external library under this specific distributive license. Libraries such as the CUDA libraries are distributed the NVIDIA Software License Agreement respectively which do not explicitly require recognition [63].

5.5 Social Issues

In regards to the social issues surrounding the project, the main issue that arises is the fact that the project is developed for the field of ariticial intelligence and machine learning. The problem arises in the fact that the misuse of data for example in training a neural network model can lead to the model having certain biases towards certain groups of people. This is a significant issue in this field as a result of the fact that if applied to real world applications, it can have profound consequences on the lives of people reliant on a service that utilises these models. This is especially true in legal cases where in utilisation of certain macahine learning models can be used to perform facial recognition on individuals without their consent. This is a significant issue as it can lead to the profiling of people without the individuals realising [83].

5.6 Ethical Issues

The ethical issues further ties in strongly with the social issues as the ethical problems are inherently implied by the connotations of the social issues. One should be ethically responsible for the data utilised when training neural network models. This is especially true if they are handling user sensitive data which under EU law is protected under the General Data Protection Regulation (GDPR) [71].

5.7 Professional Issues

Finally, in regards to the professional issues surrounding the project, the issue concerning this project arises mainly in regards to planning in terms of deadlines and budgetary requirements of the project. Mainly, the project does not inherently suffer directly as a result of budgetary constraints as the project is mainly focused on the development of software for HPC applications. However, during testing we do have to take into account that certain HPC clusters within the University of Warwick may have certain budgetary and time constraints in regards to shared wall clock compute time [68]. As an example the University of Warwick's HPC cluster within the Physics department Scientific Computing Research Technology Platform (SCRTP) has a shared wall clock time which has a budget shared by multiple departments within the University of Warwick [69]. In turn, if we did decide to make use of these HPC clusters for testing, then we must also be mindful of the budgetary constraints of the SCRTP in regards to the department paying for the compute time.

6 Conclusion

We have successfully developed a Kokkos-optimised version of the K-Nearest Neighbours (KNN) and Graph Convolutional Network (GCN) models which are able to achieve significant performance gains over the original serial implementations. We have also developed a Tensor Core-optimised version of the GCN model which is able to break the Teraflop barrier on a single Nvidia A10 GPU. We have also validated the correctness of our altered GCN model by achieving similar if not better results than the original implementation by Kipf and Welling in certain scenarios [34].

The main takeaway from this project is that the developments made by Sandia National Laboratories in the Kokkos Core library have had significant ripples in the field of high performance computing [66]. The Kokkos Core library has allowed us to pave the way for performance portable applications that can run on vast arrays of different hardware architectures optimally with minimal to no changes to the codebase.

On the other hand, Kokkos is not the be all and end all of high performance computing. In some cases, where in performance is imperative and performance portability is not a concern, then in some specific extreme cases such as high frequency trading or real-time artificial intelligence applications, then utilising hardware specialised libraries may be the optimal choice [31].
In turn, the contributions made by this project are listed as follows:

- Development of a Kokkos-optimised version of the K-Nearest Neighbours (KNN) model.
- Development of a Kokkos-optimised version of the Graph Convolutional Network (GCN) model.
- Development of a Tensor Core-optimised version of the GCN model.
- Validation of the correctness of the GCN model by achieving similar if not better results than the original implementation by Kipf and Welling [34].
- Breaking the Teraflop barrier for Full Batch GCN Training.
- Testing & Evaluation of Kokkos as a means of developing performance portable applications in the field of machine learning and artificial intelligence.

6.1 Reflection & Evaluation

The project was a significant success in terms of the development of our machine learning models using the Kokkos Core library. Additionally, the project managed to gain insight into the compute power that is available on the Nvidia A10 GPU and the performance of the Tensor Cores when applied to matrix multiplication operations. However, the project was not without its challenges. The main challenge that arose during the development of was especially time constraints with regards to the development of the Open MPI version of the application. This was as a result of lacking personal time due to commitments to other modules currently being studied during the term. Additionally, looking in hindsight of the project execution stages, the project could have been managed much more optimally with the development of the KNN model being completed in a much shorter time frame or a stepping stone algorithm to be implemented before the GCN model. This would have allowed for development of the GCN model to be much more streamlined and efficient. Perhaps allowing for time for the MPI version of the application to be developed.

6.2 Further Work

We provide two extensions to the project that could be developed in the future:

- Development of an MPI version of the application that can run on a cluster of compute nodes.
- Development of a version that supports Kokkos Kernels to allow for performance portable library performance.

The development of an MPI version of the application would allow for Full Batch GCN training to be performed on a cluster of compute nodes by breaking down each of the batches of the training set to be computed on each node. Each node would then forward propagate the data and then compute the gradients before sending the gradients to the master node to be averaged and then updated. This is effectively the same as the Kokkos-optimised version of the application but on a cluster of compute nodes utilising Kokkos with MPI [20].

This could also be combined in conjunction with a version that supports Kokkos Kernels. Kokkos Kernels is a performance portable library that allows for performance portable library optimisations to be made to the application [75]. In turn allowing for maximal performance to be achieved on a variety of different architectures within a potentially hardware diverse cluster of compute nodes.

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Andy Adinets. Introduction to cuda dynamic parallelism. `https://developer.nvidia.com/blog/introduction-cuda-dynamic-parallelism/`, May 2014.

[3] N. S. Altman. An introduction to kernel and nearest-neighbor nonparametric regression. *The American Statistician*, 46(3):175–185, 1992.

[4] AMD. Amd epyc™ 7443 spec. `https://www.amd.com/en/products/cpu/amd-epyc-7443`.

[5] AMD. Hip github repository. `https://github.com/ROCm-Developer-Tools/HIP`.

[6] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[7] T.E. Anderson, E.D. Lazowska, and H.M. Levy. The performance implications of thread management alternatives for shared-memory multiprocessors. *IEEE Transactions on Computers*, 38(12):1631–1644, 1989.

[8] David Beckingsale, Richard Hornung, Tom Scogland, and Arturo Vargas. Performance portable c++ programming with raja. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, page 455–456, New York, NY, USA, 2019. Association for Computing Machinery.

[9] L Susan Blackford, Antoine Petitet, Roldan Pozo, Karin Remington, R Clint Whaley, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, et al. An updated set of basic linear algebra subprograms (blas). *ACM Transactions on Mathematical Software*, 28(2):135–151, 2002.

[10] OpenMP Architecture Review Board. Openmp api specification for version 5.0. `https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf`, Nov 2018.

[11] Sam Buss and Alexander Knop. Strategies for stable merge sorting, 2018.

[12] Scott Chacon and Ben Straub. *Pro git*. Apress, 2014.

[13] GNU Compiler Collection. Gcc gnu documentation. `https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`.

[14] Graph500 Committee. Graph500 benchmark. `https://graph500.org/`.

[15] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[16] Crucial. Ddr4 memory specifications. `https://uk.crucial.com/support/memory-speeds-compatability`.

[17] Mehmet Deveci, Erik G. Boman, and S.Paul Rajamanickam. Sparse matrix-matrix multiplication for modern architectures. 2016.

[18] Dr. Gihan Mudalige Dr. Richard Kirk, Dr. Stephen Jarvis. Data structure abstraction and parallelisation of multi-material hydrodynamic applications. `https://wrap.warwick.ac.uk/160913/`, 2020. PhD Thesis.

[19] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202 – 3216, 2014. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[20] Message P Forum. Mpi: A message-passing interface standard. Technical report, USA, 1994.

[21] Linux Foundation. Linux kernel. `https://www.kernel.org/`.

[22] P GNU. Free software foundation. bash (3.2. 48)[unix shell program], 2007.

[23] Khronos Group. Sycl api specification for revision 7. `https://registry.khronos.org/SYCL/specs/sycl-2020/pdf/sycl-2020.pdf`, April 2023.

[24] TeX Users Group. Texworks. `https://www.tug.org/texworks/`.

[25] William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs, 2018.

[26] John T. Hancock and Taghi M. Khoshgoftaar. Survey on categorical data for neural networks. *Journal of Big Data*, 7(1):28, Apr 2020.

[27] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[28] Tomas Hrycej, Bernhard Bermeitinger, and Siegfried Handschuh. Training neural networks in single vs. double precision. In *Proceedings of the 14th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management*. SCITEPRESS - Science and Technology Publications, 2022.

[29] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open graph benchmark: Datasets for machine learning on graphs. *arXiv preprint arXiv:2005.00687*, 2020.

[30] IBM. Gradient descent. `https://www.ibm.com/topics/gradient-descent`.

[31] IBM. K-nearest neighbors algorithm. `https://www.ibm.com/topics/knn`.

[32] Intel. Intel simd architectures. `https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html`, August 2023.

[33] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[34] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks, 2017.

[35] Jon Kleinberg and Eva Tardos. *Algorithm Design*. Addison-Wesley Longman Publishing Co., Inc., USA, 2005.

[36] John C. Knight. N -version programming, January 2002.

[37] Thomas Kurbiel. Derivative of the softmax function and the categorical cross-entropy loss. `https://towardsdatascience.com/derivative-of-the-softmax-function-and-the-categorical-cross-entropy-loss-ffceefc081d`

[38] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation (OSDI'12)*, 2012.

[39] Sandia National Laboratories. Kokkos documentation. `https://kokkos.github.io/kokkos-core-wiki/`.

[40] Sandia National Laboratories. Kokkos staticcrsgraph. `https://kokkos.org/kokkos-core-wiki/API/containers/StaticCrsGraph.html`.

[41] Sandia National Laboratories. Kokkos website. `https://kokkos.org/about/`.

[42] Sandia National Laboratories. Kokkos atomics. `https://www.osti.gov/servlets/purl/1812280`, July 2020.

[43] Sandia National Laboratories. Kokkos coalesced. `https://www.osti.gov/servlets/purl/1513849`, July 2020.

[44] Sandia National Laboratories. Kokkos lecture series. `https://github.com/kokkos/kokkos-tutorials/tree/main/LectureSeries`, July 2020.

[45] Sandia National Laboratories. Kokkos parallel dispatch. `https://kokkos.org/kokkos-core-wiki/ProgrammingGuide/ParallelDispatch.html`, July 2020.

[46] Sandia National Laboratories. Kokkos tutorial 01 introduction. `https://github.com/kokkos/kokkos-tutorials/blob/main/LectureSeries/`, July 2020.

[47] Sandia National Laboratories. Kokkos tutorial 02 views and spaces. `https://github.com/kokkos/kokkos-tutorials/blob/main/LectureSeries/`, July 2020.

[48] Oak Ridge National Laboratory. Frontier supercomputer debuts as world's fastest, breaking exascale barrier. *Oak Ridge National Laboratory*, May 2022.

[49] R.S. Latha, G.R. Sreekanth, R.C. Suganthe, M. Geetha, R. Esakki Selvaraj, S. Balaji, K.R. Harini, and P. Priya Ponnusamy. Stock movement prediction using knn machine learning algorithm. In *2022 International Conference on Computer Communication and Informatics (ICCCI)*, pages 1–5, 2022.

[50] Daniel Lustig and Margaret Martonosi. Reducing gpu offload latency via fine-grained cpu-gpu synchronization. *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 354–365, 2013.

[51] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S. Vetter. Nvidia tensor core programmability, performance mixed precision. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, May 2018.

[52] Matthew Martineau, Simon McIntosh-Smith, and Wayne Gaudin. Assessing the performance portability of modern parallel programming models using tealeaf. *Concurrency and Computation: Practice and Experience*, 29(15):e4117, 2017. e4117 cpe.4117.

[53] Microsoft. Microsoft outlook. `https://www.microsoft.com/en-gb/microsoft-365/outlook/email-and-calendar-software-microsoft-outlook`.

[54] George Millington. Amd expands professional offerings with amd radeon™ pro vii workstation graphics card and amd radeon™ pro software updates. *AMD Press Release*, 2020-05-13.

[55] M. E. J. Newman. The structure and function of complex networks. *SIAM Review*, 45(2):167–256, 2003.

[56] NVIDIA. cublas documentation. `https://docs.nvidia.com/cuda/cublas/`.

[57] NVIDIA. Cuda c programming guide. `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`.

[58] NVIDIA. cudnn documentation. `https://docs.nvidia.com/deeplearning/cudnn/index.html`.

[59] NVIDIA. Nvidia nsight compute. `https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html`.

[60] Nvidia. Cuda memory management. `https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html`, August 2023.

[61] Nvidia. Cuda toolkit documentation 12.2 update 2. `https://docs.nvidia.com/cuda/`, August 2023.

[62] Nvidia. Nvidia a10 datasheet. `https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a10/pdf/a10-datasheet.pdf`, August 2023.

[63] Nvidia. Nvidia software license agreement. `https://www.nvidia.com/en-gb/drivers/nvidia-license/`, 2023.

[64] Nvidia. Thrust documentation. `https://docs.nvidia.com/cuda/thrust/index.html`, August 2023.

[65] Nvidia. Thrust sort documentation. `https://docs.nvidia.com/cuda/thrust/index.html#sorting`, August 2023.

[66] US Department of Energy. Exascale computing project. `https://www.exascaleproject.org/research`.

[67] University of Warwick. Tabula. `https://tabula.warwick.ac.uk/`.

[68] University of Warwick. Hpc midlands+. `https://warwick.ac.uk/research/rtp/sc/hpc/hpcmidplus/`, May 2023.

[69] University of Warwick. Research technology platforms. `https://warwick.ac.uk/research/rtp/sc/`, April 2024.

[70] University of Warwick Department of Computer Science. Warwick batch system. `https://warwick.ac.uk/fac/sci/dcs/intranet/user_guide/batch_compute/`.

[71] Information Commissioner's Office. Overview of the general data protection regulation (gdpr). https://ico.org.uk/for-organisations/data-protection-and-the-eu/overview-data-protection-and-the-eu/#:~:text=in%20the%20EEA.-,Does%20the%20GDPR%20still%20apply%3F,law%20as%20the%20UK%20GDPR., August 2021.

[72] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019.

[73] S.J. Pennycook, J.D. Sewall, and V.W. Lee. Implications of a metric for performance portability. *Future Generation Computer Systems*, 92:947–958, 2019.

[74] Bertrand Putigny, Brice Goglin, and Denis Barthou. A benchmark-based performance model for memory-bound hpc applications. In *2014 International Conference on High Performance Computing and Simulation (HPCS)*, pages 943–950, 2014.

[75] Sivasankaran Rajamanickam, Seher Acer, Luc Berger-Vergiat, Vinh Dang, Nathan Ellingwood, Evan Harvey, Brian Kelley, Christian R. Trott, Jeremiah Wilke, and Ichitaro Yamazaki. Kokkos kernels: Performance portable sparse/dense linear algebra and graph kernels, 2021.

[76] Sebastian Ruder. An overview of gradient descent optimization algorithms, 2016.

[77] T. Konstantin Rusch, Michael M. Bronstein, and Siddhartha Mishra. A survey on oversmoothing in graph neural networks, 2023.

[78] Victor Sanchez. Learning theory. https://warwick.ac.uk/fac/sci/dcs/teaching/material/cs342/lecture4_learningtheory.pdf.

[79] Oak Ridge National Laboratory Sandia National Laboratories. Kokkos tech conference 2017. https://www.osti.gov/servlets/purl/1525572, May 2017.

[80] SBIR. Sbir funding for kokkos tensor library. https://www.sbir.gov/node/2232575.

[81] Bernhard Schölkopf, Alexander Smola, and Klaus-Robert Müller. Kernel principal component analysis. In Wulfram Gerstner, Alain Germond, Martin Hasler, and Jean-Daniel Nicoud, editors, *Artificial Neural Networks — ICANN'97*, pages 583–588, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[82] Amazon Web Services. Amazon robotics case study. https://aws.amazon.com/solutions/case-studies/amazon-robotics-case-study/, August 2021.

[83] Amazon Web Services. Amazon rekqognition. https://aws.amazon.com/rekognition/, 2023.

[84] William Stallings. *Computer Organization and Architecture: Designing for Performance*. Prentice Hall Press, USA, 8th edition, 2009.

[85] Copyright (2020) National Technology and LLC (NTESS). Engineering Solutions of Sandia. Kokkos usage rights. https://kokkos.github.io/kokkos-core-wiki/license.html.

[86] PROMETEUS Professor Meuer Technologieberatung und Services GmbH. Top500 statistics. `https://www.top500.org/statistics/overtime/`.

[87] Cornell University. K-nearest neighbors algorithm. `https://www.cs.cornell.edu/courses/cs4780/2018fa/lectures/lecturenote02_kNN.html`.

[88] Valgrind. Valgrind. `https://valgrind.org/`.

[89] Guido Van Rossum and Fred L Drake Jr. *Python reference manual*. Centrum voor Wiskunde en Informatica Amsterdam, 1995.

[90] Zeke Wang, Hongjing Huang, Jie Zhang, and Gustavo Alonso. Benchmarking high bandwidth memory on fpgas, 2020.

[91] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. A comprehensive survey on graph neural networks. 2019.

[92] Weiren Yu. Activation functions, 2023-2024. Available at `https://warwick.ac.uk/fac/sci/dcs/teaching/material/cs331/cs331-copy-22-23/2022_lecture_13_activation_functions_2.pdf`.

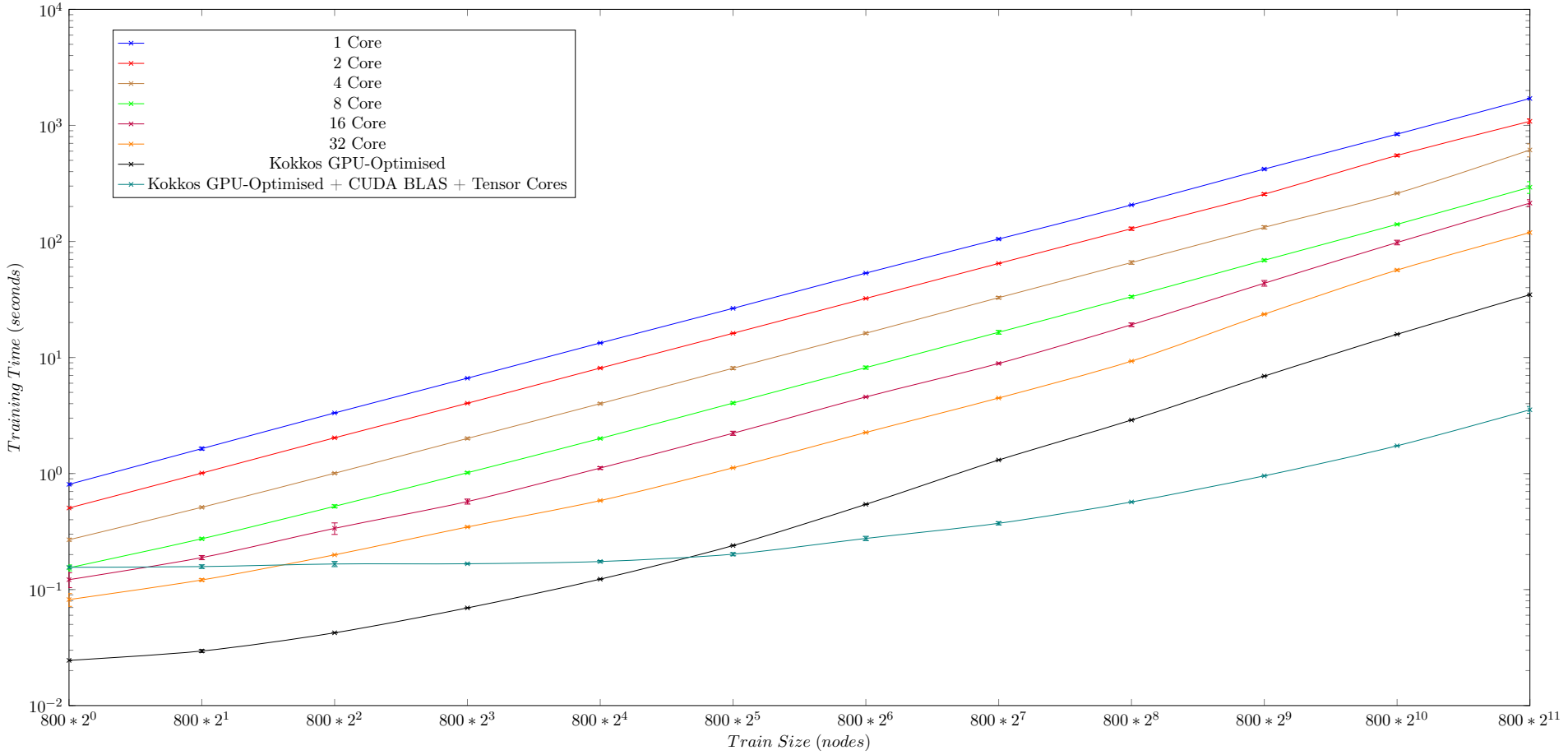
[93] Weiren Yu. Backpropagation, 2023-2024. Available at `https://warwick.ac.uk/fac/sci/dcs/teaching/material/cs331/cs331-copy-22-23/pdf_2022_lecture_17_backprop.pdf`.

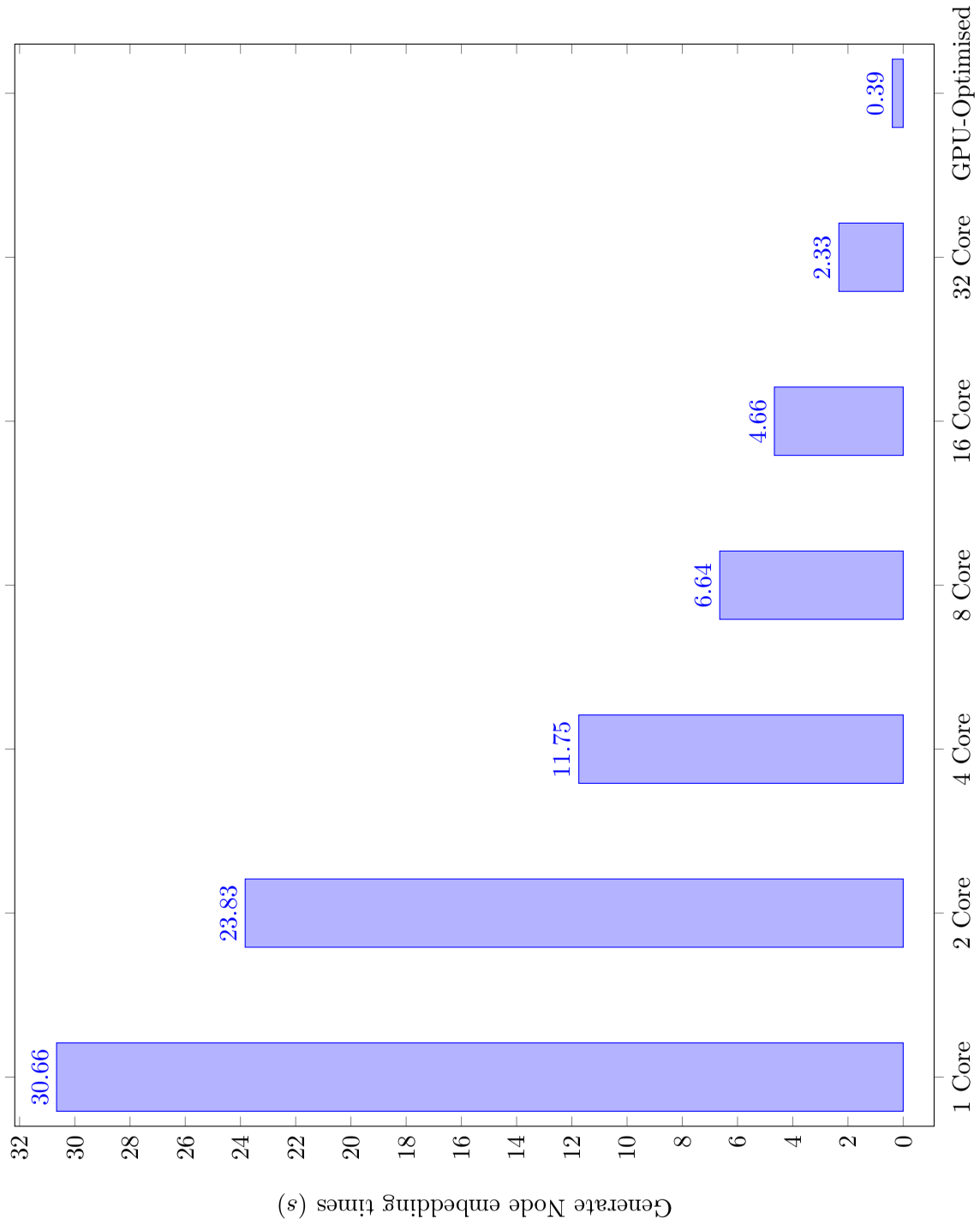
[94] Weiren Yu. Loss function and regularisation, 2023-2024. Available at `https://warwick.ac.uk/fac/sci/dcs/teaching/material/cs331/cs331-copy-22-23/pdf_lecture_16_loss_function_and_regularisation.pdf`.

[95] Si Zhang, Hanghang Tong, Jiejun Xu, and Ross Maciejewski. Graph convolutional networks: a comprehensive review. *Computational Social Networks*, 6(1):11, Nov 2019.

A Appendix

A.1 Previous Results




Generate Node embedding times (s)
0
2
4
6
8
10
12
14
16
18
20
22
24
26
28
30
32
30.66
23.83
11.75
6.64
4.66
2.33
0.39
1 Core
2 Core
4 Core
8 Core
16 Core
32 Core
GPU-Optimised

A.2 Previous Gantt Chart