Distributed Matrix Multiplication using Raspberry Pis

Ryan Wiebe

Master of Science

Computer Science

School of Informatics

University of Edinburgh

2024

Abstract

The rapid growth of the Internet of Things has led to machine learning being increasingly used for safety-critical applications such as autonomous vehicles. The current paradigm of offloading computationally expensive calculations from edge devices, to remote high-performance computing clusters has shown to be incapable of meeting hard real-time constraints. This is because these clusters are often distant from edge devices, leading to a large communication latency. One promising alternative to address this latency is to offload these computations to more numerous lower-cost edge servers, which are proximate to the edge devices.

This work explores the viability of an alternative approach that offloads these calculations directly to large clusters of edge devices instead of more costly edge servers. This has been done by developing a simplistic primary-secondary distributed framework that offloads matrix multiplication tasks to numerous Raspberry Pi 5s. Matrix multiplication was selected for offloading as it forms the foundation of many contemporary machine learning algorithms. If suitable scalability can be demonstrated with matrix multiplication using Raspberry Pi 5s, it is likely that more complex machine learning algorithms can also be scaled efficiently.

Two experiments were conducted to determine whether matrix multiplication tasks can be suitably scaled when using the distributed framework on a single and multiple Raspberry Pis. Results demonstrated near-linear scalability when using up to three Raspberry Pis, with the primary being the bottleneck, meaning the implemented framework scales well.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Ryan Wiebe*)

Acknowledgements

I wish to dedicate this work to the memory of my uncle, Neil Harte, who recently passed away. His guidance was invaluable in shaping me into the person I am today.

Foremost, I would like to express my deepest gratitude to my supervisor, Dr. Yuvraj Patel, for his insightful feedback and guidance. His encouragement and expertise was vital in allowing me to succeed.

Additionally, I would like to thank my colleagues Gerardo Melesio Sancen, Hanyuan Ma, and He-Yi Lin for their advice and camaraderie, which was critical in helping me tackle challenges as they appeared.

Finally, I would like to express appreciation to my partner, my parents, and my friends for their patience and unwavering support. You all have been a key source of strength and encouragement throughout this journey.

Table of Contents

Chapter 1

Introduction

1.1 Background

The rapid growth of the Internet of Things (IoT) has led to machine learning (ML) systems being increasingly used for applications with hard real-time components [1]. It is becoming more apparent that utilising remote high performance computing (HPC) clusters for offloading ML inference is not able to meet the latency required for many systems, such as those that are safety critical (i.e. object detection in self-driving cars) [2, 1]. With there being an expected 29.42 billion IoT devices online by 2030 [3], alternative approaches must be developed to facilitate ML inference with low latency.

Though much research has already been conducted to explore potential approaches that could accelerate ML calculation speed and the memory usage on IoT *edge devices* through software optimisation and additional hardware support [4, 5, 6, 7], these approaches are still limited by monetary cost (i.e. cost for additional required hardware) and power cost (i.e. additional read/write operations) related to using a single device for increasingly large calculations. In addition, on-device techniques for inference, such as pruning for neural network compression, can lead to a loss in accuracy [8, 9], which is not necessarily desired in safety-critical applications (i.e. autonomous vehicles).

A distributed approach that offloads these calculations to numerous lower-costing *edge servers* that are proximate to the IoT devices has been shown to be a good alternative [10, 11]. Our work attempts to take this to the extreme by exploring the viability of directly offloading these complex calculations directly to a cluster of system-on-chip-powered (SoC) edge devices. While much work has been done in distributing ML algorithms on low-cost edge devices [12, 13], these works tend to be primarily concerned with distributing the calculations without fully addressing the scalability

to a large number of devices. If comparable ML inference performance could be demonstrated when using multiple edge devices while maintaining a similar or lower cost, clusters of these low-cost devices could be dispersed at more locations that are closer to IoT devices. This would effectively allow ML computation offloading at a low latency without sacrificing computation power. We believe this hypothesis may be the case, as work such as [12] have already demonstrated promising improvements in terms of inference time when using clusters of SoC-powered devices.

1.2 Objectives

Our work is exploratory in nature and serves as an initial validation of the viability of distributing a basic ML algorithm directly on edge devices before exploring more complex use cases. This project aims to demonstrate this by showing matrix multiplication can be scalably distributed across numerous edge devices. We have chosen to use matrix multiplication as the target algorithm as it is the basis of many more complex ML algorithms such as neural networks, linear regressors, and kernel functions. If such scalability is observed in distributing this fundamental calculation, similar behaviour may be possible with more sophisticated algorithms. Our work targets a cluster of Raspberry Pi (RPI) 5s due to their low cost and widespread usage in both industry and academia. We have set the following three objectives for this project:

1. Develop a suitable framework that can distribute matrix multiplication across multiple RPIs.

2. Evaluate whether using a single RPI for distributed matrix multiple is scalable (i.e. does using multiple cores scale).

3. Determine whether suitable scalability is observed when using multiple RPIs.

We have chosen to design and implement a new RPI-specific distributed matrix multiplication framework, as few currently available frameworks are suitable for our needs. Traditional frameworks such as Dask [14] and Apache Spark [15] are designed primarily for large HPC clusters without considering the limited computing capabilities that devices such as the RPI have. While some RPI-specific distributed frameworks exist [16, 17], they tend to be designed for generic use cases that are not necessarily optimised for distributing matrix multiplication.

Chapter 2

Literature review

This chapter outlines the findings gained throughout a literature review. The research has focused on matrix multiplication algorithms, distributed ML, and contemporary distributed matrix multiplication systems.

2.1 Matrix multiplication algorithms

2.1.1 Conventional matrix multiplication

Matrix multiplication is the process of multiplying two input matrices to produce a new matrix. Given two input matrix A and B where A is a matrix of $\mathbb{R}^{m \times n}$ and B is a matrix of $\mathbb{R}^{n \times p}$, the matrix multiplication of $A * B$ will result in matrix C, which is $\mathbb{R}^{m \times p}$ [18]. As seen in figure 2.1, matrix multiplication is conventionally performed using the sum of element-wise multiplication across each row of A and each column of B [18]. This approach is computationally complex in the sense it requires n^3 multiplications and $n^2 * (n-1)$ additions to generate a result [18].

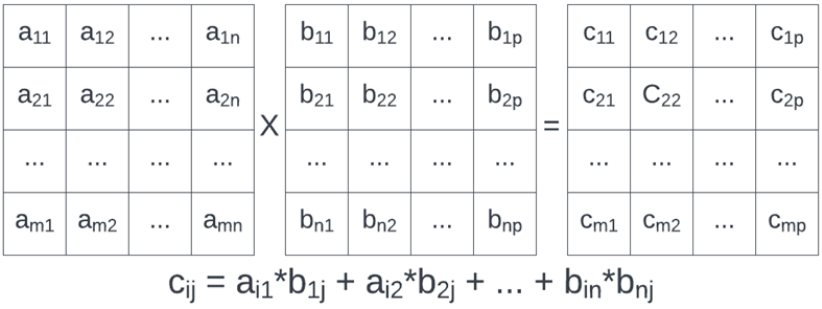

Figure 2.1: Diagram demonstrating how matrices A and B can be multiplied together using conventional matrix multiplication.

One challenge when implementing conventional multiplication algorithms on computers is the order in which elements inside matrices A and B are accessed. As seen in figure 2.2, three algorithms for this are the *row-by-column*, *row-by-row*, and *column-by-column* [18, 19]. Row-by-column matrix multiplication effectively iterates through each row of A and multiplies it with all elements in B by iterating along B's columns. Alternatively, row-by-row matrix multiplication instead performs multiplication by iterating through B's rows instead of columns. Column-by-column operates similar to row-by-column but instead iterates through A along columns instead of rows. [19] conducted an experiment to determine which of these three algorithms has the fastest execution time. It found row-by-row to offer the best average execution time. This is likely due to processors favouring sequential memory accesses due to caching. Matrices are typically stored in a 1D sequential array in memory, with each row being stored sequentially (i.e. the first row occupies the first n entries in the 1D array, assuming a matrix of size $n \times n$). Iterating along columns effectively causes each memory access to request locations with a n difference in addresses. Assuming n is larger than a cache line, cache misses are likely to occur with each column iteration.

Row-by-Column

```
for ( i = 0; i < size; i++)
  for ( j = 0; j < size; j++)
    for ( k = 0; k < size; k++)
      A[i, j] += B[i, k] * C[k, j]
```

Row-by-Row

```
for ( i = 0; i < size; i++)
  for ( j = 0; j < size; j++)
    for ( k = 0; k < size; k++)
      A[i, k] += B[i, j] * C[j, k]
```

Column-by-Column

```
for ( i = 0; i < size; i++)
  for ( j = 0; j < size; j++)
    for ( k = 0; k < size; k++)
      [k,i] += B[k,j] * C[j,i]
```

Figure 2.2: Pseudo code of row-by-column, row-by-row and column-by-column element-wise algorithms as described in [18, 19].

2.1.2 Strassen's matrix multiplication

In modern processors, addition operations are typically faster than multiplication operations. Strassen's algorithm attempts to take advantage of this by utilising fewer multiplications and more additions when multiplying two matrices [20]. Consider the 2×2 matrices in figure 2.3, Strassen's algorithm can multiply these two matrices using seven multiplications and 18 additions/subtractions instead of eight multiplications and four additions when using the conventional algorithm [18]. This corresponds to approximately $\sim n^{2.807}$ multiplications as opposed to the conventional's n^3[18].

Strassen's algorithm can be used on larger matrices using recursion. Consider the multiplication of matrices A and B, which are of size $n \times n$. We recursively divide the matrix into half-sized block matrices and apply Strassen's algorithm at a certain

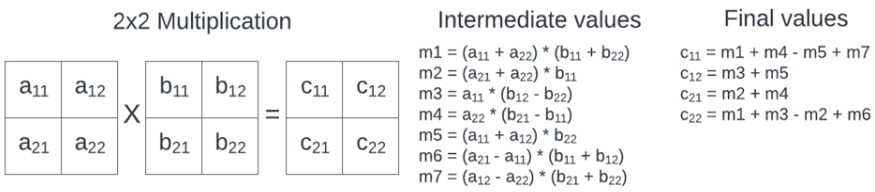

Figure 2.3: Diagram demonstrating the various intermediate values used in Strassen's 2×2 matrix multiplication algorithm.

cutoff matrix size. Assuming $n = 2^q$, we can expect this recursion to continue until all multiplications are between individual elements as opposed to blocks of matrices. If this is not the case, conventional matrix multiplication algorithms can be applied at the set cut-off size.

2.2 Distributed machine learning

Before analysing distributed matrix multiplication algorithms, it's important to understand distributed ML as a whole. Distributed ML systems operate by splitting a large ML computation into smaller computations, which can be concurrently executed using multiple devices called *nodes*. There are many topologies that can be used when arranging the communication and computation operations of these nodes. Common topologies include primary-secondary systems (i.e. all nodes feed computation results into a single primary node), tree-based systems (i.e. each node controls some set of child nodes, which they can further offload computation to), and decentralised systems (i.e. each node operates independently with no form of centralised control) [21].

Two common ways of achieving parallelism are *data parallelism* (i.e. partition the dataset itself across multiple nodes that each contain the same model) and *model parallelism* (i.e. partition the model itself across multiple nodes to accelerate time to compute a single sample) [22]. Assuming each node contains the same computation capabilities, work must be evenly distributed across the available nodes to maximise the benefits of parallelism. Our work is primary concerned with model parallelism as we wish to accelerate a single matrix multiplication calculation using multiple computation nodes.

Two approaches that are common in frameworks for distributed ML are *MapReduce* (MR) and *Bulk-Synchronous Processing* (BSP) [22]. BSP works by splitting an algorithm into a set of parallelisable operations called supersteps [23]. Parallelisable operations are executed concurrently on various nodes. Note that each superstep must

be executed in sequence, meaning straggling nodes could affect performance. MR is a more simplistic model that breaks an algorithm into a series of map and reduce phases [24]. The map phase first executes a map function that assigns keys to local data stored on each node. A shuffle phase then occurs where data of specific keys is redistributed across nodes such that all data of a set key is located at a single node. The reduce phase then occurs, where each node takes the data it received during the shuffle phase and reduces it to a final result. One benefit of MR's simplistic model is that it can be scaled easily across a large number of nodes [21].

2.3 Distributed matrix multiplication

Distributed matrix multiplication is a form of model parallelism that distributes matrix multiplication computation work using a set of concurrent nodes to generate a single result matrix. Key factors in distributing matrix multiplications include reducing redundant operations (i.e. avoid unnecessary calculations) and redundant communications (i.e. unnecessary inter-node communication steps) [25]. Two methods of dividing matrix multiplication work across nodes include *non-block* and *block-based approaches* [26].

2.3.1 Non-block matrix multiplication

Non-block approaches take advantage of the independent nature of the multiplications within conventional matrix multiplication algorithms. These approaches will tend to distribute a series of rows or columns of matrix A to each node for local storage. Each node will typically then continuously perform multiplications across all of matrix B until completion [27]. [27] conducted experiments on the performance of various non-block approaches that distribute row/columns across a number of CPU cores. While [27] showed that more execution cores led to a near-linear decrease in calculation time, it is questionable whether an approach like this is scalable to a truly distributed system with multiple nodes as opposed to a single node with multiple execution cores. Each node will effectively either be required to store a full copy of matrix B, which could lead to a computation bottleneck, or will be required to receive B matrix data through communication, leading to large communication costs. Due to this, the vast majority of literature is focused on block-based approaches.

2.3.2 Block-based approaches

Block-based approaches work by dividing the two matrices A and B into a series of block submatrices, which are used for computation across each node. Block-based literature typically assumes a fully decentralised model with nodes arranged into a grid/cube that operate using a BSP-like system. Nodes are able to communicate with their neighbouring nodes (i.e. a given node can communicate with neighbours above, below, to the left, and to the right). Matrices A and B are typically divided in N blocks for a grid of NxN nodes. Each node is typically able to store one block from matrices A, B, and a special block to store local partial results. Optimisation typically occurs by minimising the communication required between nodes.

The foundation of block-based distributed matrix multiplication is Cannon's algorithm [28]. As seen in figure 2.4, each node initially stores blocks from matrix A and B corresponding to the node's position in the grid (i.e. p_{11} initially contains block A_{11} and B_{11}). Each node then calculates the matrix multiplication of the two locally stored blocks. Once all nodes have performed their local multiplications and added the result to it's local partial result matrix, they each send their local A matrix to their left neighbour (i.e. left shift) and their local B matrix to their above neighbour (i.e. up shift). Assuming an NxN grid of nodes, this process is repeated N times, where the partial result stored in each node will correspond to the matrix multiplication result for its given block. Further improvements have been made since Cannon's algorithm through communication optimisation, additional local memory usage on each node, and the use of Strassen's algorithm [29, 30, 31, 32, 33, 34, 35]. [33, 34, 35] are particularly notable as they demonstrated distributed matrix multiplication can be thought of as a tree of multiplication operations on submatrices of a set size. We use a similar approach when generating computation tasks in our system (see section 3.4.2).

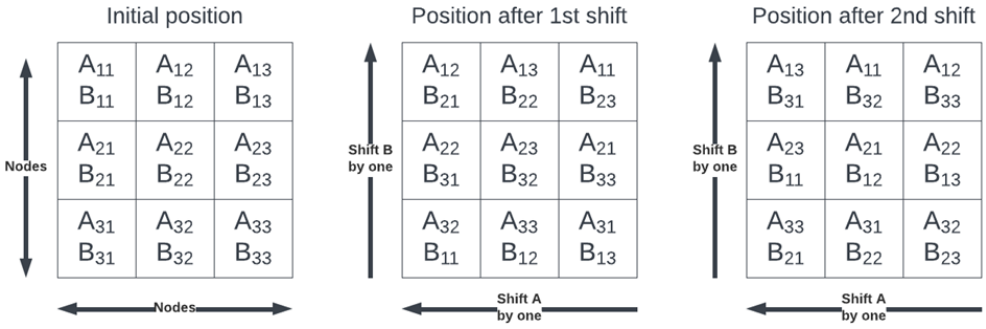

Figure 2.4: Diagram showing the distribution of data across the processor grid when computing the multiplication of 3×3 matrices using Cannon's algorithm.

Chapter 3

Implementation

This chapter offers a high-level description of the implemented distributed matrix multiplication system. The system utilises a primary-secondary paradigm where the matrix multiplication of two large matrices is divided into a tree made of discrete matrix multiplication and matrix multiplication tasks that can be offloaded from a primary to numerous secondary RPI 5 devices. This work has been implemented primarily in C with a C++ interface for interacting with gRPC, which has been used for primary-secondary communication.

3.1 Raspberry Pi 5 theoretical hardware capabilities

A model of the RPI 5 was initially constructed to determine the theoretical hardware capabilities before designing and implementing a system. As seen in figure 3.1, the RPI 5 utilises four ARM Cortex A76 cores inside the BCM2712 system on chip as its primary processors [36]. Each Cortex A76 core contains a 64kB write-back L1 cache and a 512kB dynamic-biased L2 cache, which both utilise 64B cache lines and the MESI algorithm for cache coherency [37]. The four cores also share a 2 MB L3 cache [36]. Additionally, the RPI 5 offers high speed communication through two USB 3.0 ports and a 1 Gb/s Ethernet port [36].

In a simplistic distributed matrix model, the RPI must receive some data, perform some computations, and output the result. An ideal system would have a communication speed that is significantly faster than its computation speed to ensure minimum computation downtime. A brief investigation was conducted to compare the RPI's multiplication theoretical execution speed to its theoretical communication speed. The Cortex A76 has a signed integer *multiply and accumulate* (MnA) instruction and *add*

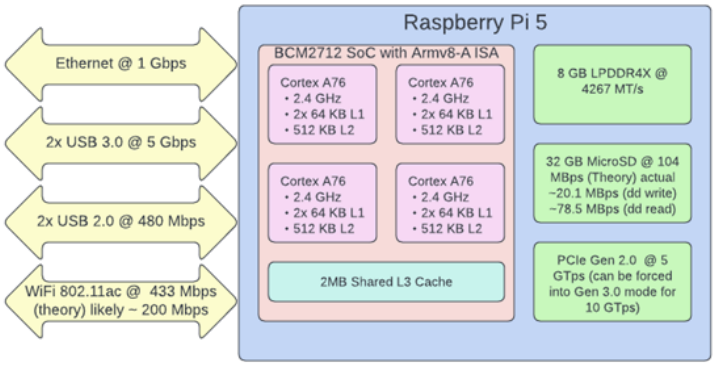

Figure 3.1: Basic block representation of RPI hardware include the BCM2712 system on chip and various input/output interfaces.

instruction throughput of 1 instruction/cycle and 3 instructions/cycle, respectively [38]. Assuming no load instructions and a steady clock speed of 2.4 GHz, we could expect 2.4 G MnA ops/s (as three multiplications are required for every N in conventional matrix multiplication, the practical MnA rate is 0.8 G MnA ops/s) and 7.2 G add ops/s.

With the Cortex A76 supporting out-of-order operations, costs due to the RPI's operating systems, and this estimation not accounting for time required for memory accesses, we can't expect this performance in reality. The matrix multiplication and addition algorithms, as discussed in section 3.4.1, were run on a single Cortex A76 core to determine the true rate at which input data could be consumed when performing matrix multiplication and matrix addition. This experiment was conducted using buffers, which correspond to the total memory at the L1-L3 cache sizes, to minimise bottlenecks due to cache misses. As each MnA instruction takes two 4 byte signed integers as inputs in the Cortex A76 [39], it was found that the core was consuming input data at rates of $\sim$11.8 MB/s ($\sim 68\times$ slower than ideal), $\sim$4.2 MB/s, and $\sim$2.2 MB/s for L1, L2, and L3 data sizes. As the theoretical output speed of the Ethernet is 125 MB/s, the system's bottleneck during matrix multiplication will likely be the computation time, even in instances where all cores are executing multiplication, assuming no computation cost for Ethernet communication. Despite this, matrix addition consumed inputs at a higher rate of $\sim$1.1 GB/s ($\sim 6.5\times$ slower than ideal) at all cache sizes, meaning a network bottleneck is likely present when performing addition.

3.2 Primary-secondary paradigm and assumptions

Chapter 2 showed that current block-based distributed matrix multiplication systems utilise a local storage model with direct point-to-point communication between nodes. While these approaches offer great performance, they are complex, and their performance can be affected by numerous implementation-specific factors. As discussed in 1.2, the work in this paper is exploratory and is to serve as the basis for future work, hence, we have opted to use a simplistic distributed matrix multiplication system using a basic primary-secondary paradigm. The primary benefit of this is it allows us to isolate all factors external to the RPI and solely focus on the scalability of on-device multiplication itself in ideal circumstances.

Our primary-secondary approach assumes there exists a primary device with infinite computation capabilities that has a point-to-point communication link with an arbitrary number of RPI secondary devices. The primary is given oracle access to the two input matrices but is forbidden from performing any computations directly related to the matrix multiplication. Instead, it must offload these computations to secondary devices by dividing the total multiplication work into a series of smaller subcomponents called tasks. Each task contains two input matrices and an operation to perform. Additionally, each task must be executable by only a single secondary. The secondary simply receives the input matrices from a single task and returns an output matrix to the primary after executing the requested operation. For reasons discussed in section 3.4.2, these operations will either be matrix multiplication or matrix addition. Once all tasks have been executed, the matrix multiplication has been complete. As the tasks are independent and the secondaries are homogeneous RPIs, we can expect task throughput to increase linearly for every added RPI. If we can demonstrate this linear behaviour, we can then assume that similar scalability may be possible with more complex algorithms, such as those outlined in section 2.3.2.

3.3 High-level software design

Two applications primarily written in C were created to represent the primary and secondary devices discussed in section 3.2. gRPC via the RPI's Ethernet was chosen for primary-secondary communication as: 1) the 1 Gbps Ethernet interface should provide sufficient bandwidth to prevent a communication bottleneck on a single RPI when performing matrix multiplication (see section 3.1), and 2) gRPC is a lightweight library

that uses a simplistic client-server communication model via TCP. While alternative libraries such as MPICH were investigated [40], they are bare bones and require the development of additional mechanisms such as a load balancer. While implementing such low-level network-specific features would likely be beneficial for performance, this work is to serve as the baseline implementation for future projects, hence, implementing these is outside the scope of this work.

In short, the primary device has three primary purposes:

1. To divide the large input matrix into smaller tasks, which can each be executed by a secondary RPI.

2. To ensure each secondary RPI has been allocated and is executing a set number of tasks.

3. To manage which idle tasks are ready for subsequent execution.

The secondary serves two purposes:

1. To execute received tasks evenly across a set number of cores on the RPI.

2. To communicate with the primary by receiving input task data and returning the corresponding output data.

One way of representing the software implementation is in a series of producer-consumer problems across 6 distinct software layers - the top three run in the primary's application and the bottom three run in the secondary's application. The first layer is made up of the read and write threads, which are responsible for allocating tasks to the RPIs. The second layer is made up of the *DistMultClient* class, which is responsible for tracking the status of this work and initiating gRPC communication. The third and fourth layers are responsible for transferring data along the Ethernet wire. The fifth layer is made up of the *DistMultServer* class, which is responsible for passing received work to RPI cores and sending result data back via gRPC. The sixth layer is made up of 1-4 core-pinned threads, which are responsible for executing the originally allocated work.

The interaction between these layers when executing a single task has been outlined in figure 3.3. The primary first selects a secondary to send a task to. The input is then encapsulated using a DistMultClient object and sent to the secondary via Ethernet. The secondary then receives the task inside its DistMultServer object and passes it to a core-pinned thread for execution. Upon completion of this execution, the inverse

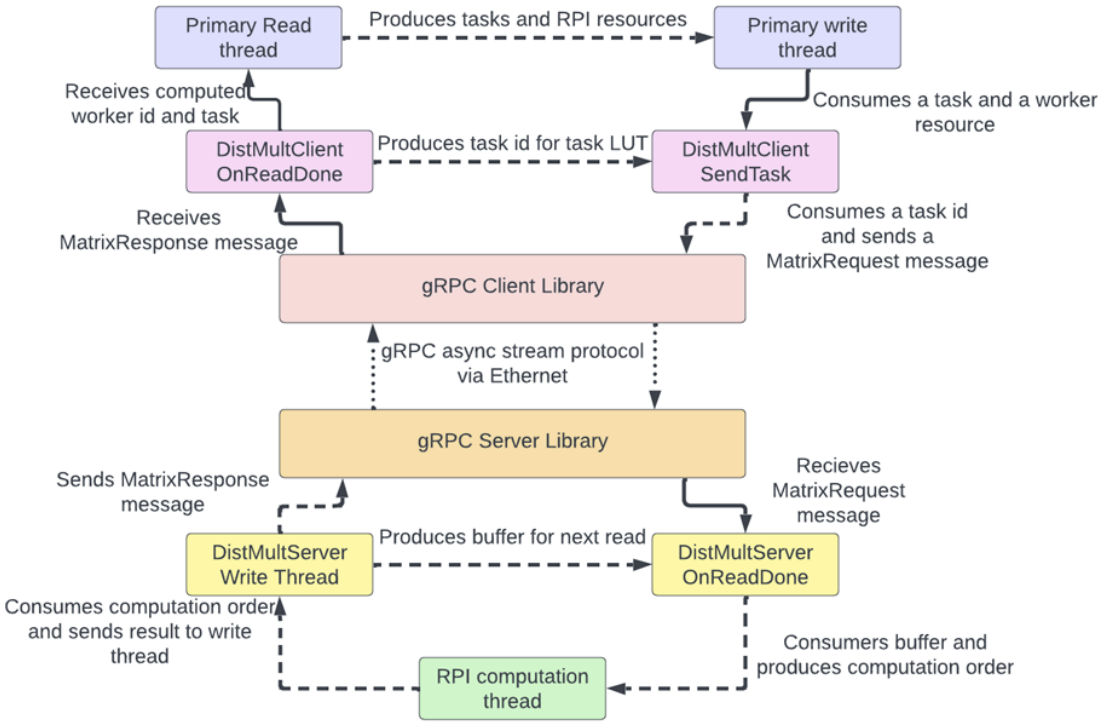

Figure 3.2: Diagram showing the six software layers in the software architecture. Note the various producer-consumer paradigms at each layer.

of this operation then occurs, with the secondary encapsulating the result using its DistMultServer object and returning it to the primary.

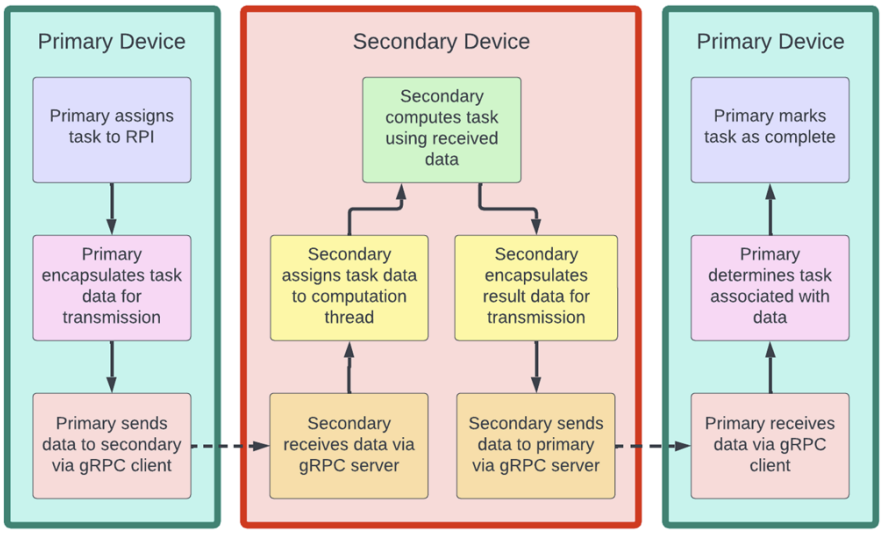

Figure 3.3: Diagram of the lifetime of a task as it flows between the primary to a secondary devices.

There are two effective modes of operations when assigning tasks to a given secondary: *synchronous* and *asynchronous*. Synchronous works by having the primary only assign a single task to each computation thread. The primary first sends this task to the secondary. The secondary then receives the task, computes it inside a computation thread, and returns it back to the primary. Upon receiving the task result, the primary then sends the next task. This operation is not ideal as each computation thread effectively is left idle while data is being sent to and from the primary. Asynchronous addresses this by instead assigning two tasks to each computation thread. Each computation thread computes one of these tasks while receiving another task from the primary in the background. Given the computation time will likely be the bottleneck due to reasons discussed in section 3.1, we can assume the buffers associated with the second task will be full before the computation thread finishes executing the first task, allowing the computation thread to continuously execute tasks with minimal computation downtime. It is recommended to use the representation of software layers in figure 3.2 and the task lifetime in figure 3.3 as a reference when reading the rest of this chapter.

3.4 Implementation preliminaries

Before discussing the implementation of the primary and secondary devices, the following subsections outline any features on which the wider implementation is reliant.

3.4.1 Matrix representations

Typically, $m \times n$ arrays (i.e. matrices) in C are represented on the stack by unwrapping each n-sized row into a 1D array of size $m \times n$. Column-wise iteration is handled by offsetting the memory address by n. This is not necessarily the case for the heap, meaning a suitable heap-based representation must be created. Two different matrix representations were developed utilising arrays of signed integers with different use cases. As seen in figure 3.4, these are the *1D* and *2D* representations.

The 2D matrix representation uses an ”array of arrays” approach inside a structure called *matrix_t*. Say we had an $m \times n$ matrix, an array of signed integer pointers will be allocated on the heap to size m. Each of the pointers in this array will point to heap-allocated integer arrays of size n. This representation is not cache-friendly as each allocated array will not necessarily be located at continuous addresses, meaning many cache misses will likely occur when iterating from the end of one row to the start

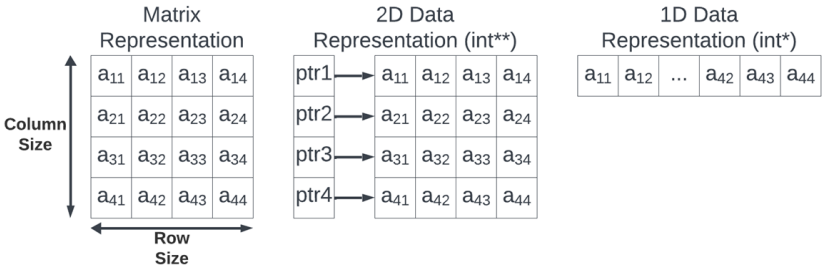

Figure 3.4: Diagram showing how the 1D and 2D matrix approaches are used to represent a 4×4 matrix.

of another. Despite this drawback, this representation is very flexible for generating submatrices with direct memory pointers to the original matrix. Due to this, it is used by the primary when it generates a task tree and in it's task representation. This can be seen in figure 3.9 in section 3.4.2.

The 1D representation is similar to stack-based multi-dimensional arrays in the sense it uses a single allocated integer array of size $m \times n$. The primary benefit of this continuous memory approach is that it is very cache-friendly. As the memory addresses are continuous, subsequent rows will likely already be pre-loaded into cache when iterating from the end of one row to the start of another. Not only is this favourable when iterating through the matrix for computations, data received on a network link will naturally be in a 1D representation and can be directly passed into a 1D matrix structure without the need for extra copying. These traits have led it to be primarily used by the secondary device for performing all matrix computations. See sections 3.5.3 and 3.5.4 for more on how the secondary uses the 1D representation. Note that the 1D representation is encapsulated inside a structure called *matrix_1d_t*.

A simple cache-friendly matrix multiplication algorithm was implemented to ensure efficient computations occur using the 1D representation on the secondary. As discussed in section 2.1.1, the row-by-row conventional matrix multiplication algorithm offers the best performance when compared to it's counterparts due to it's inherent cache-friendly behaviour. It has hence been implemented to perform all multiplications with the 1D matrix representation. A simple matrix addition algorithm was also implemented by simply iterating through both the left and right matrices and performing element-wise addition.

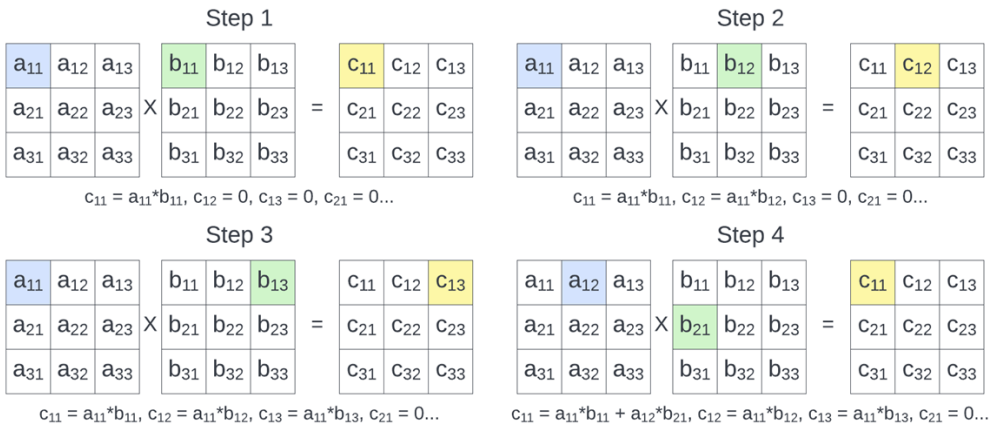

Figure 3.5: Diagram demonstrating the row-by-row multiplication algorithm from [18, 19]. The algorithm works by multiplying each value in an row of A by it's corresponding row in B and storing the partial result (i.e. A1 multiplied by B's first row, A2 by B's second row, etc.).

3.4.2 Tasks and task tree creation

Before discussing the various layers, it's important to understand what tasks are and how they can be used to perform distributed matrix multiplication. The proposed system divides the multiplication of two large matrices into a series of smaller computations called tasks. As demonstrated in [34, 33], we can think of matrix multiplication as a series of ordered calculations that can be arranged into a series of tree data structures, with each node representing a computation of some form with a result that is passed to another node. As seen in figure 3.6, we opted to use a similar, more simplistic approach that decomposes a matrix into multiple same-sized blocks and creates a tree of multiplications and additions between these blocks based on the conventional matrix multiplication algorithm. Each leaf node in the tree represents a piece of input data, with the root node being the final output data. Each corresponding input data is linked together using a corresponding multiplication task, which generates a result. Multiplication task results are then passed into corresponding addition tasks until a final result is generated. The tree shown in figure 3.6 shows the graph for cell c_{11}. Similar trees must be created to represent each output cell.

The primary uses a very similar approach and generates these *task trees* by performing multiplication and addition operations on blocks of submatrices instead of individual matrix cells. As seen in figure 3.7, the primary first divides its input matrices into smaller submatrix blocks of a set size. In the case of figure 3.7, it divides the 4×4 matrix into four $2x2$ submatrix blocks. Using these submatrices, task trees are

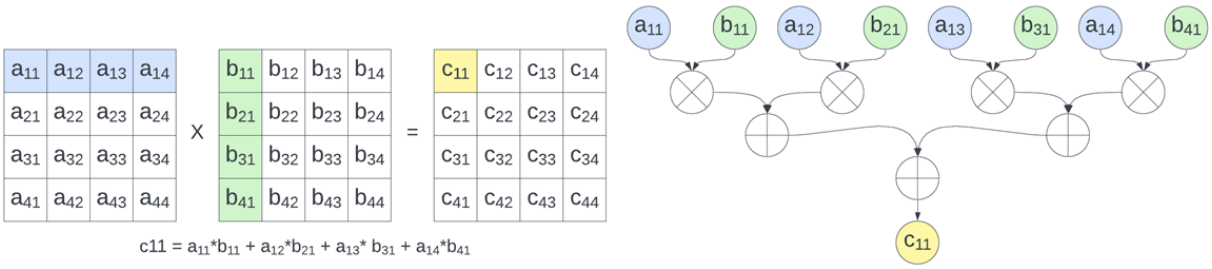

Figure 3.6: Diagram showing how the calculation of an output cell using conventional matrix multiplication algorithm can be represented as graph of multiplication and addition tasks. The coloured nodes represent the input and output data from each respective matrix. The white nodes represent the multiplication and addition tasks.

constructed for the result for each of the output's submatrices. Note that the number of task trees to generate corresponds to the total number of output submatrices (i.e. one task tree per submatrix). By offloading the computation of all tasks in each task tree to secondaries, distributed matrix multiplication can occur. Note that the implemented system can only generate task trees for square matrix multiplication (i.e. number of rows equals the number of columns) with square submatrices. Additionally, similar task trees can be implemented for other matrix multiplication algorithms, such as Strassen's algorithm. Due to time constraints, we have only implemented task trees for conventional matrix multiplication.

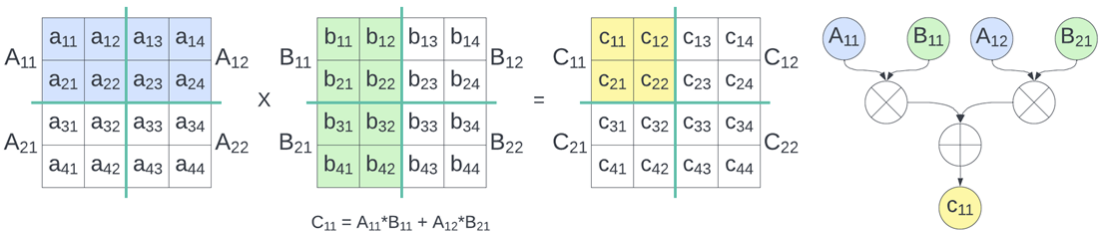

Figure 3.7: Diagram showing how the primary divides a matrix into block submatrices and generates a task tree which can be computed to generate a given output block's value. The coloured nodes represent the input and output data from each respective matrix. The white nodes represent the multiplication and addition tasks.

One challenge with task trees is when the number of multiplications or multiplications in a layer of the task tree is not a power of 2. As seen in figure 3.8, the result of an excess multiplication task cannot be paired with another multiplication task for addition. When this occurs, a new addition task is created and appended to the root of the task tree. While doing this could lead to a bottleneck forming towards the root of the tree when

using multiple secondaries to execute a single task tree. This isn't an issue in practice as tasks from all trees are simultaneously executed using a first-in-first-out (FIFO) queue (see section 3.5.1 for more on this). Assuming the number of trees generated is greater than the number of available secondaries, no bottleneck should form.

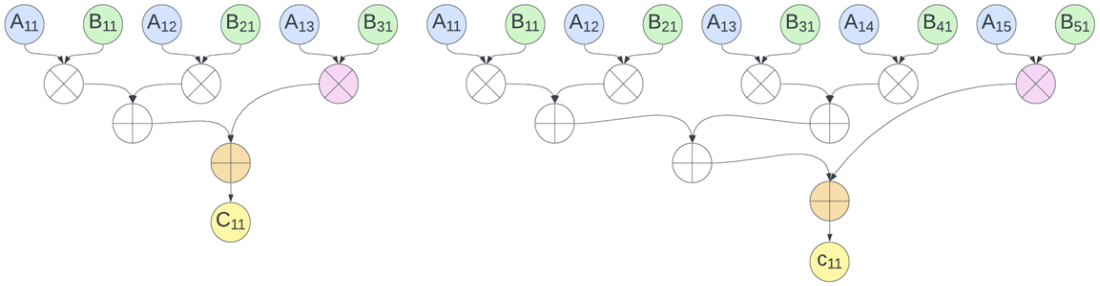

Figure 3.8: Diagram showing how excess tasks are handled in tree construction in 3x3 and 5x5 matrix multiplications. The excess tasks which cannot be paired with another task in it's layer has been highlighted in purple. The addition task appended at the root of the tree is highlighted in orange.

As seen in figure 3.9, the data structure used to represent each task on the primary is *task_node_t*. It contains information about the operation it represents in the form of an enumeration that determines whether the task is for matrix addition or matrix multiplication, an unsigned int corresponding to the id of the RPI currently executing the task (see section 3.5.1 for more on this), two input 2D matrix structures, and a single 2D matrix result structure. In order to represent the order of the tree, each task_node_t structure also contains a pointer to its parent, left child, and right child task_node_t structures. These pointers are null if the corresponding node is a leaf node or parent node.

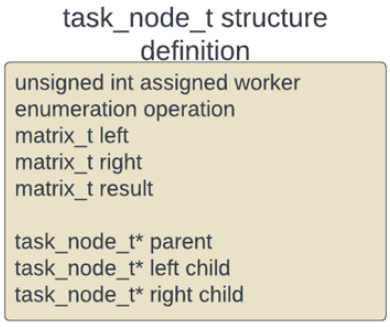

Figure 3.9: Definition of task_node_t structure. This is used by the primary to represent tasks.

3.4.3 Primary-secondary communication

As described in section 3.3, gRPC is used to facilitate primary-secondary communication across the RPI's 1 Gb/s Ethernet port. gRPC communication operates through definable services, which contain various methods that a gRPC client can trigger on a gRPC server for computation [41]. Our system utilises a service called *DistMultService* which runs on every RPI using gRPC's internal server library. The primary can interact with a given secondary's gRPC server by running its own client instance using gRPC's internal client library. These internal libraries represent layers 4 and 3, respectively.

Communication between the client and service is done using DistMultService's only callable method called *ComputeMatrix*, which has an input stream of *MatrixRequest* messages and an output stream of *MatrixResponse*. Asynchronous streams in gRPC are a one way method of continuously sending or receiving data on the client or server without blocking [41]. Each stream contains two methods that the client and server can use: write and read operations [41]. Write operations are used to send data from the client to the server or vice versa [41]. Read operations are used to receive incoming data from the client/server [41].

There are two message types that are used in these streams, each represented by C++ classes. The MatrixRequest class is made up of four unsigned integers and two byte arrays. The unsigned integers contain information such as the type of operation to perform (i.e. multiplication or addition), a task id, the row size of the two matrices, and the column size of the two matrices. The byte arrays correspond to the 1D representation (i.e. matrix_1d_t structure) of the left and right input matrices of a task. These messages are sent using the input stream from the client (i.e. primary) to the server (i.e. secondary). Similarly, the MatrixResponse class contains the task id, column and row sizes, and a byte array for the 1D representation of the result matrix. These messages are used to send data from the server to the client.

The DistMultClient (layer 2) and DistMultServer classes (layer 5) were created to manage and interface all communications between the client and server of the gRPC service. Each class effectively contains a member function to write data and a member function called OnReadDone, which is called internally by gRPC when a piece of data is read. As layers 1 and 6 were written in C, a C-based interface was also created to instantiate, call member functions, and destroy these objects as needed. Note that a separate DistMultClient must be created for each DistMultClient running on a secondary, meaning N client objects will be created for N secondary devices.

3.4.4 Task scheduler

One challenge common to the primary and secondary is balanced task allocation between *execution units*. An execution unit is something that can consume multiple tasks simultaneously. This task allocation challenge has been addressed by implementing a hierarchical linked list with each entry representing an execution unit. Each entry is made up of an id (i.e. which execution unit the entry corresponds to) and a resource counter (i.e. how many tasks can the execution unit handle simultaneously). This id is used by the primary/secondary to pass said task to the execution unit. See sections 3.5.1 and 3.5.3 for more on how this id is used. The maximum value of the resource counter can be set as needed.

A component called *task_scheduler_t* was implemented to represent this hierarchical linked list. The order of these entries in the list is descending based on the value of each entry's resource counter, with the entry with the highest counter being at the start of the link list. There are two primary operations that can be performed on the linked list: 1) consume a resource from the entry with the most resources, and 2) produce a resource on a selected entry. Note that each operation is mutex-protected to ensure thread safety.

The resource consumption operation is used to gather the id of the next execution unit to pass a task to. As seen in figure 3.10, resource consumption works by saving the id of the linked list's head entry and decrementing the entry's resource counter. Next, the linked list is updated by moving the previous entry to a new position based on its new resource value. The id is finally returned. If an entry in the linked list has zero resources available after consumption, it is removed from the linked list completely. If a resource consumption operation occurs and there are no entries in the linked list, the operation will wait until an entry has been added.

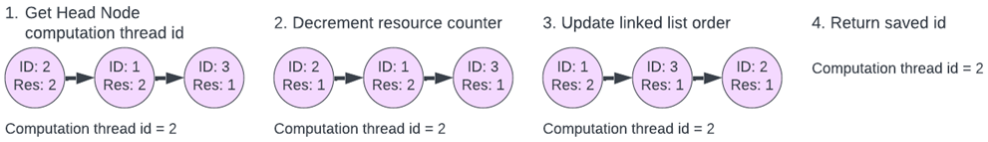

Figure 3.10: Diagram showing how consumption operations are performed on a task_scheduler_t to gather the id for the next computation thread buffer to read into.

The resource production operation works by first incrementing the resource counter of the entry associated with a received computation thread id. It then reorders the linked list to ensure the updated entry is at the correct position. If the entry was not in the linked list as it had zero resources, it is added to the tail of the linked list.

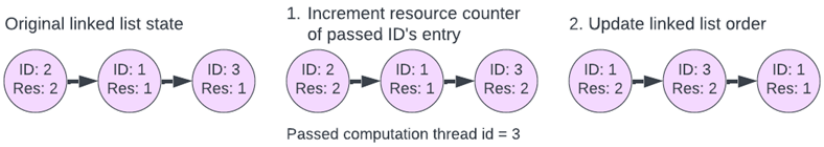

Figure 3.11: Diagram showing how production operations are performed on a task_scheduler_t.

3.4.5 First-in-first-out queue

Another challenge common to the primary and secondary is passing ordered data between threads. This has been handled by implementing a FIFO queue in a dedicated structure called queue_t. This structure contains a linked list of entries with the head node corresponding to the first-in entry. Each queue_t can have either push or pop operations performed on it. The push operation works by creating a new node, storing a pointer to some data, and then appending it to the end of the linked list. The pop operation simply removes the head node and returns the pointer assigned to the original head node. Both operations have been mutex-protected to ensure thread safety.

3.5 Task execution

3.5.1 Task and secondary allocation

As discussed in section 3.3, the primary must assign an idle task to a secondary device to begin execution. This can effectively be thought of as a producer and consumer problem in regards to executable tasks and the RPIs to allocate said tasks to. The layer's implementation is made up of two threads: the write thread (i.e. consumes tasks and RPIs) and the read thread (produces tasks and RPIs).

Task production and consumption works using a queue_t, with the write thread pulling tasks from the queue and the read thread pushing tasks to it. The queue effectively contains all tasks that are leaf nodes in every single task tree. This is because these nodes have all of their input data assigned to a buffer and hence can be computed. The queue's initial state contains all multiplication leaf nodes of every created task tree. As seen in figure 3.13, the write thread simply gathers the head of the queue to consume a resource. If no tasks are in the queue, the write thread will wait for a task to be produced by the read thread.

One challenge with handling multiple RPIs is that gRPC requires that each gRPC

server connection have a DistMultClient. This means *N* client objects are required to communicate with *N* RPIs. As seen in figure 3.12, this has been handled by using a simple look-up table (LUT) of DistMultClient objects with each entry corresponding to an RPI. Each LUT index corresponds to an RPI's id. Tasks are sent from the primary to one of the RPIs by selecting an id and using the corresponding DistMultClient object.

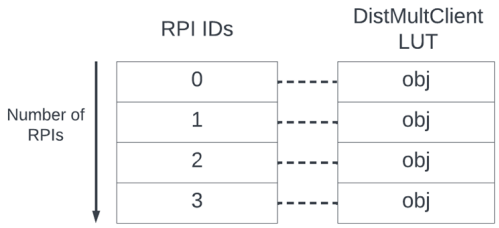

Figure 3.12: Diagram showing the relationship between an RPI id and the DistMultClient objects inside the primary's gRPC LUT.

Ideally, tasks should be assigned evenly across the RPIs. This means id selection should account for the number of tasks currently assigned to each RPI. This has been handled using task_scheduler_t, with each entry corresponding to an RPI's id from the LUT and the number of resources corresponding to the number of tasks that can be assigned to each RPI. The number of tasks per RPI depends on whether synchronous or asynchronous operation is used. In synchronous, each of the RPI's computation threads can handle a single task. This means the number of tasks that can be assigned to a single RPI is the number of computation threads (i.e. four tasks per RPI when four computation threads per RPI). In asynchronous, each computation thread can handle two tasks at a time (i.e. one for computation and one to be filled via background communication). This means the number of tasks per RPI is two times the number of computation threads (i.e. eight tasks per RPI when four computation threads per RPI). To send a task to an RPI, the resource consumption operation outlined in section 3.4.4 is performed on the task_scheduler_t, and an id is returned. Task data is then sent using the DistMultClient object corresponding to the returned id. As seen in figure 3.13, the write thread will effectively continue to send tasks to the RPI with the most available resources (i.e. the least amount of assigned tasks) until there are no available RPI resources or available task resources. Assuming there is no task bottleneck and the time to receive the first result from an RPI is longer than the total allocation time, the distribution of work sent to RPIs should be balanced.

As seen in figure 3.13, the read thread is reliant on messages sent to its inbound queue_t from each RPI's DistMultClient object. When a DistMultClient object receives

the output of a task, the read thread is first sent the id of the RPI that has finished executing a task. This causes the read thread to free up a resource used by the RPI's entry in the task_scheduler_t by using the produce operation outlined in section 3.4.4. The read thread then receives the completed task, causing it to potentially produce a new task using a task clean-up operation. This operation removes the now computed task from the task tree and passes a pointer of the result data to the input of it's parent node. If the parent node has input data from both children after passing the pointer, a new task is ready to be computed and is pushed to the back of the task queue.

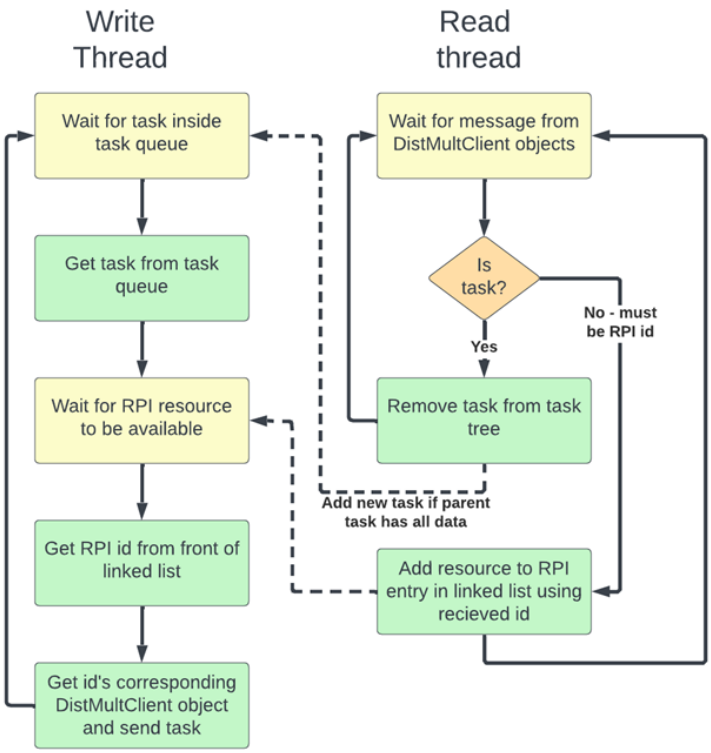

Figure 3.13: Flow chart of interaction between the read and write thread. Dotted lines represent creation of additional shared resources.

3.5.2 Primary gRPC interface

As discussed in section 3.3, the DistMultClient class is used as an interface between the read/write threads and the internal gRPC client library. This class has two primary purposes: 1) to send data from a task to the object inside the write thread using write operations, and 2) to notify the read thread of data received for the corresponding RPI using read operations.

The DistMultClient effectively operates as a simple extension of the producer-

consumer paradigm between the read and write threads. Each DistMultClient object must be able to handle at least the maximum number of tasks that can be assigned to it's corresponding RPI. As each task may be executed on different cores on the RPI, there is no guarantee these tasks will be returned in the order they are sent. Due to this, an LUT of tasks with an associated task id has been introduced (see figure 3.14). This task id has a producer-consumer paradigm where sending a message consumes an LUT slot and receiving a response produces a new LUT slot using a C++ queue of task ids. When the write thread sends a message via gRPC, it calls the SendTask DistMultClient member function and passes a task as an argument. This task is assigned a task id from the task id queue, causing a pointer to the task to be stored in the task LUT. All required data is then copied to a MatrixRequest message, including the task id, which will be later returned in the corresponding MatrixResponse message. This message is then passed into the internal gRPC client library using a write operation, where it will eventually be sent, allowing the write thread to continue it's iteration. Note that there could be a significant delay from the time of passing the message to the internal gRPC library to when the message is actually sent through Ethernet.

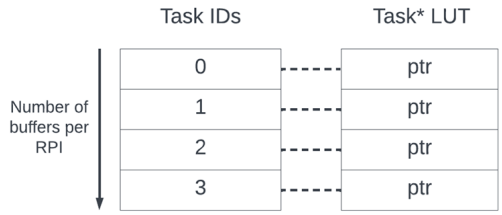

Figure 3.14: Diagram showing the relationship between a task id and the task buffer used by each DistMultClient object.

The DistMultClient object contains a callback function called OnReadDone, which is called from the internal gRPC library when a MatrixResponse message is received from the RPI during a read operation. When called, the DistMultClient first sends the it's associated RPI id to the read thread. This allows the read thread to update the number of resources the RPI has available, hence allowing the next write to begin without delay. Next, the task id is gathered from the MatrixResponse object. Using this task id, the task corresponding to this message is gathered from the task LUT. The task id is then added to the end of the DistMultClient's task id queue so it can be consumed for additional sends. The 1D matrix data received inside the MatrixResponse object is then copied into the 2D matrix buffer of the task. The task is finally sent to the reader thread so additional tasks can be produced.

3.5.3 Secondary gRPC interface

As discussed in section 3.3, each RPI has a DistMultServer object, which is responsible for interacting between the computation threads and the internal gRPC server library. Upon receiving data inside a MatrixRequest object, all necessary data is offloaded and passed to a computation thread inside a *task_compute_data_t*. This contains all the data required for matrix computations, including two input matrix_1d_t structures, an enumeration value determining whether a multiplication or addition operation is to occur, and a matrix_1d_t structure to store the output. It similarly passes result data in the form of a task_compute_result_t from each computation thread and sends it to the primary. The definition of these can be seen in figure 3.15.

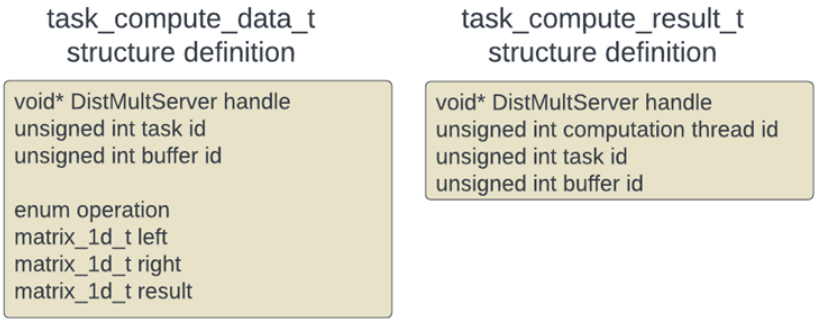

Figure 3.15: Definition of task_compute_data_t and task_compute_result_t structures. This is used by the secondary's computation threads to receive and send messages.

One challenge posed by using gRPC is that the byte arrays inside both MatrixRequest and MatrixResponse are represented by C++ strings. Ideally, we want each computation thread to have its own set of buffers to benefit from cache locality. Due to this, a method is required to directly map these C++ buffers to each computation thread. This has been handled by assigning each computation thread a set number of stack-allocated MatrixRequest objects (used for reading inbound messages), task_compute_data_t structures (used to store messages to computation threads), MatrixResponse objects (used to send data), and the computation thread queue_t structure (used for writing outbound messages) inside a number of single/multi-dimensional LUTs. As discussed in section 3.3, the number of tasks assigned to each computation thread depends on whether synchronous or asynchronous operation is used. If synchronous operation is used, the LUT is single dimensional, with each computation thread only having a single MatrixRequest, task_compute_data_t and MatrixResponse associated with it, however, if asynchronous operation is used, each computation thread instead has two of each buffer type.

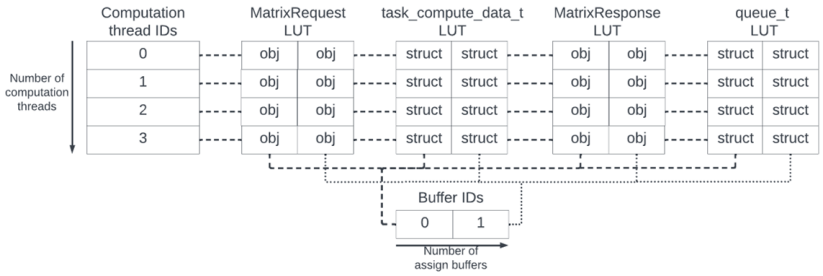

Figure 3.16: Diagram showing the relationship between the computation thread and buffer ids and the buffers inside the secondary's LUTs.

Ideally, task execution should be spread evenly across an RPI's computation threads. One challenge with implementing this is that gRPC requires a MatrixRequest object to be specified before beginning the next read. As each buffer is directly associated with a computation thread, a buffer management system was implemented to select which computation thread will compute the next inbound task. This was done by tracking the number of tasks allocated to each computation thread using a task_scheduler_t, with each entry corresponding to a computation thread and each resource counter corresponding to the number of available MatrixRequest objects. Using a consumption operation, a computation thread is selected, and the next read will feed into it's corresponding MatrixRequest object in the LUT. As asynchronous assigns two MatrixRequest objects per thread, the least recently used MatrixRequest object will be used for the next read.

A flow chart of the read and write operations inside the DistMultServer class can be seen in figure 3.17. When the DistMultServer is reading, a MatrixRequest object from the LUTs is used, with its corresponding computation thread id being saved. When OnReadDone is called, this stored id is used to determine which MatrixRequest object contains the incoming data. A new read is then potentially initiated using another LUT buffer if a computation thread has an available resource. Next, critical information, including the buffer id, the operation to perform, and the matrix dimensions, are stored inside the matrix_compute_data_t in the LUT slot with the same buffer id. Direct pointers for the buffers from the MatrixRequest and MatrixResponse with the same buffer id are then passed to the matrix_compute_data_t's input and output matrix_1d_t structures. This is possible as the data format used for the inbound matrix buffers is already in the 1D representation. Not only does doing pointer copying prevent the need for an additional copy from one buffer to another, it also offers cache benefits as the computation thread will likely already have the cache lines from each buffer in memory, allowing for write-back benefits from the Cortex A76's MESI cache coherency protocol.

The matrix_compute_data_t is then sent to the corresponding computation thread using the queue_t with the same buffer id.

Write operations have been handled by using a separate thread internal to the DistMultServer class. It operates by simply waiting for task_computation_result_t structures to be pushed into it's queue_t. Upon receiving a structure, the computation and buffer ids are used to determine which MatrixResponse is ready to be sent. Basic information included in the task_computation_result_t such as the task id are then attached to this MatrixResponse object. Note that no memory copies are necessary as the computational thread directly stores the computation results into this object's result buffer. If no MatrixRequest buffer is currently being used for reading, the least-recently used MatrixRequest object is then used to start a new read. If a MatrixRequest buffer is already being used for reading, a resource is instead freed using the received computation thread id.

Figure 3.17: Flow chart demonstrating how read and write operations occur within DistMultServer. Note that DistMultServer performs writes within a dedicated write thread and performs reads inside it's OnReadDone callback function which is controlled by a thread inside the internal gRPC server library.

3.5.4 Secondary task execution

As discussed in section 3.3, each RPI contains a number of computation threads that are used to perform matrix multiplication and addition tasks. As seen in figure 3.18, these threads operate primarily using two queue_t structures as input and output. Each thread initially waits for its input queue to for a task_compute_data_t structure to be received from the gRPC interface. Upon receiving the task_compute_data_t structure, the input matrix_1d_t structures are then used to either perform matrix multiplication or matrix addition using the 1D matrix algorithms described in section 3.4.1 depending on the value of the received enumeration. Once computed, the result is stored inside a stack-allocated task_compute_result_t structure, which is sent to the DistMultServer's output queue. The thread then waits idle until another input matrix is received.

As discussed in section 3.1, the RPI contains four Cortex A76 cores, meaning the maximum number of computation threads is four, with each thread being pinned to a single core of the RPI. This is done to prevent the operating system's kernel from unnecessarily interacting with computation threads, causing task computation delays and cache flushing. The number of computation threads used is varied at compile time using compiler tags.

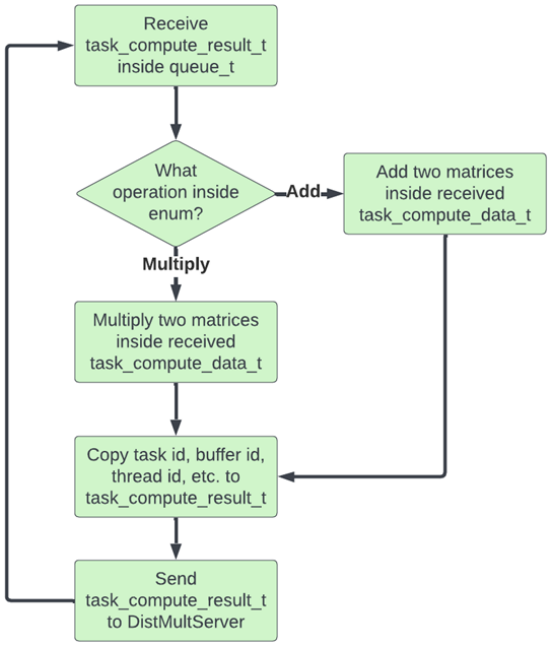

Figure 3.18: Diagram showing the flow of execution used by the secondary's computation threads.

Chapter 4

Evaluation

This chapter outlines the testing and evaluation conducted on the implemented distributed matrix multiplication system. This includes a description of methods used to validate system behaviour. In addition, two different experiments have been outlined in this chapter to analyse single device scalability and multi device scalability of the synchronous and asynchronous RPI strategies.

4.1 System validation

Before conducting the experiments outlined in section 4.2, the implementation outlined in section 3 was validated. The primary method of system validation was log analysis of primary and secondary device operation, and a use of the gdb debugger. System performance was analysed using these as additional features were added. In addition, numerous assertions were placed throughout the code to catch invalid states as they occur, such as an assertion to ensure RPI actually has resources before sending a task or that the final multiplication result generated by the system was correct. The system was able to successfully compute the matrix multiplication of two $11,800 \times 11,800$ matrices using a submatrix size of 295×295 using three RPIs each with four computation threads. This corresponds to $64,000$ matrix multiplication and $62,400$ addition tasks successfully completed without an assertion arising. Additionally, no assertion arose due to invalid states, nor did the system fail to generate a correct result when computing this large matrix and the various simulations described in sections 4.5 and 4.6. These two facts are indicative of the implemented system's robustness.

4.2 Evaluation methodology

Three metrics have been primarily used to evaluate the performance of the synchronous and asynchronous multiplication systems. These are:

1. Total clock time of completing all tasks (*Total clock time*).
2. Total clock time of completing all multiplication tasks (*Total mult time*).
3. Total clock time of completing all addition tasks (*Total add time*).

The total clock time is measured from when the write thread on the primary sends it's first task to a DistMultClient object to when the read thread receives it's last task. Similarly, the total multiplication/addition clock time is measured from when the first multiply/addition task is sent to when the final multiplication/addition task is received by the read thread. Note that the sum of the total multiplication time and the total addition time does not necessarily equal the total time taken as an add task may be sent to an RPI before the final multiplication task has been handled by the primary. Changes in clock time should be indicative of the system's scalability.

We are also interested in identifying challenges in system scalability; hence, a model of synchronous and asynchronous tasks was created, with the various associated time costs being measured. As seen in figure 4.1, the theoretical time taken for each synchronous task from the secondary computation thread's perspective is the time spent performing the required matrix addition/multiplication (i.e. *mult cost* and *add cost*) added with the time spent sending/receiving data (i.e. *comm cost*). In reality, there is an additional time cost (i.e. *addi cost*) that occurs on the secondary when transferring data between the DistMultServer object and a computation thread. This is the time spent transferring data from DistMultServer's OnReadDone function to a computation thread (i.e. setup time), the time a data_compute_result_t spends waiting inside DistMultClient's output queue for handling, and the time spent passing received data_compute_result_t into a MatrixResponse before beginning a gRPC write.

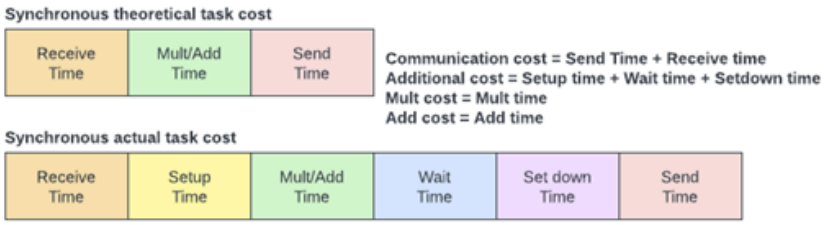

Figure 4.1: Diagram of the theoretical and actual time costs with synchronous tasks.

Asynchronous operates by filling a second set of task buffers in the background of another task's matrix multiplication/addition computation; hence, the theoretical time taken for each asynchronous task from the secondary computation thread's perspective is the time taken for each matrix multiplication/addition. In reality, there will likely be time spent waiting in between finishing the one task computation and receiving the next task on the computation thread's input queue. This time must be added to the mult/add time (i.e. *idle cost*).

Three further metrics were gathered to evaluate whether communication or computation is a bottleneck when performing matrix multiplication tasks:

1. Multiplication input data rate (i.e. *Mult data rate*).
2. Communication data rate (i.e. *Comm data rate*).
3. Effective data rate (i.e. *Eff data rate*).

Multiplication input data rate is the rate at which input data is consumed during matrix multiplication. This is calculated by taking the total size of a single task's input matrices in bytes and dividing it by the average time taken for each task's on-device matrix multiplication. This can be seen in Equation 1, where n^2 is the row size of each of the task's square input matrix, t_m is the average time spent performing matrix multiplication, N_{rpi} is the number of RPIs, and N_{thread} is the number of computation threads per RPI. Note that n is multiplied by a factor of two as there are two input matrices and a factor of four due to each signed integer being 4 bytes on the Cortex A76 [39].

$$\text{Equation 1: } Mult\ data\ rate = N_{rpi} * N_{thread} * (2 * 4 * n^2)/t_m$$

The effective communication data rate is the rate at which data is effectively transmitted via gRPC. This is calculated by taking the total amount of bytes sent per task and dividing it by the average communication cost per task. This can be seen in Equation 2, where t_{comm} is the average communication cost. Note that n^2 is multiplied by a factor of three as three matrices are sent during communication and a factor of four due to each signed integer being 4 bytes on the Cortex a76 [39].

$$\text{Equation 2: } Comm\ data\ rate = (3 * 4 * n^2)/t_{comm}$$

The effective data rate is the effective rate at which input data is consumed when executing each multiplication task. This is calculated by taking the total size of a single

task's input matrices in bytes and multiplying it by the number of multiplication tasks, and then dividing it by the total multiplication time. This can be seen in Equation 3, where N_{tasks} is the number of multiplication tasks computed and T_{mult} is the total mult time.

Equation 3: $\textit{Eff data rate} = N_{tasks} * (2 * 4 * n^2)/T_{mult}$

4.3 Metric gathering

A metric gathering system was implemented to record the metrics outlined in section 4.2 from each distributed matrix multiplication run. This has been done by using Linux's internal *clock_gettime* function, which can be used to determine elapsed time at nanosecond resolution [42]. As some time was tracked on the secondary's side, this information was passed back inside MatrixResponse messages in the form of doubles. Given the matrix sizes used for evaluation were in the magnitude of kBs, these additional bytes should have minimal effect on the overall communication time. Additionally, *RunStats* and *RpiStats* C++ classes were implemented to track each desired metric. RunStats is primarily used to track the system's overall times while RpiStats is primarily used to track the times associated with each individual RPI (i.e. each DistMultClient object contains it's own RpiStats object where it places various timings for that specific RPI upon receiving data inside the client's OnReadDone function). Data gathered by these classes can be exported to a CSV file if needed upon completing all matrix multiplication tasks.

4.4 Evaluation limitations

The main limitation of the experiments outlined in sections 4.5 and 4.6 is the performance of the primary. As discussed in section 3.2, we assume the primary has infinite computation capabilities and communication bandwidth. This is not true for the experiments, as a laptop with a 6-core AMD Ryzen 5 7640U and a 2.5 Gbps Ethernet link was used as the primary. Given that the purpose of this work was to focus on the scalability of the RPI itself, the primary's code implementation is not necessarily optimal and could cause further bottlenecks. Both of these factors effectively mean the scalability of the primary will play a factor as the number of computation threads and RPIs is varied.

4.5 Single device scalability

The objective of this experiment was to determine whether the synchronous and asynchronous systems were scalable as the number of computation threads was varied on the secondary. As seen in figure 4.2, the experiment was set up by running the secondary application on a single RPI 5 and connecting it to the primary via a 1 Gb/s Ethernet link. This experiment entailed running various matrix multiplication tasks, measuring the various total times, and analysing whether linear improvements were observed as the number of computation threads was varied from one to four (i.e. one for each Cortex A76 core). To ensure a fair test, the experiment was run with submatrix sizes, which caused the total buffer sizes used by each RPI core to be at approximately L1, L2, and L3 cache sizes (i.e. the sum of matrix A, B, and C buffers for synchronous, and the sum of matrix A, B, and C buffers multiplied by a factor of two for asynchronous). This corresponds to submatrix sizes of 70×70, 208×208, and 410×410 for synchronous, and 52×52, 147×147, and 295×295 for asynchronous. Not only does using these submatrix aid in minimising cache misses, it also gives an indication of whether there is an ideal submatrix size for both systems. The outer matrix size used for the experiment was chosen to be five times the chosen submatrix size (i.e. a 350x350 outer matrix is used for a 70x70 submatrix). This means each experiment will execute 125 multiplication tasks and 100 addition tasks at the given submatrix size.

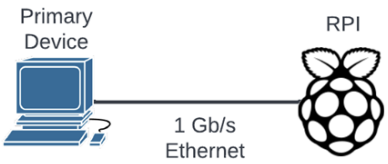

Figure 4.2: Diagram of single device experiment.

This section will offer an overview of key aspects from the results, however, the full set of results is available in Appendix A. Tables 4.1 and 4.2 display the total times and task costs from executing at 410x410 and 295x295 submatrix sizes using the synchronous and asynchronous strategies, respectively. Both approaches saw total multiplication time decrease by $\sim 2\times$ and $\sim 3\times$ when using two and three threads, respectively, implying near-perfect scalability. Despite this, there is a drop off to $\simeq 3.75\times$ when using four threads. This drop off is likely due to the computation thread sharing a core with the gRPC. Not only does this cause numerous context switches due to the gRPC and computation thread sharing execution time, these context switches can also cause an increase in cache misses due to gRPC and the computation thread

likely working with different data. Interestingly, the total add time did not decrease linearly with additional threads for any submatrix size. This is because the matrix addition time is significantly shorter than the communication cost for a given matrix size, meaning little benefit is offered from additional threads as the Ethernet's bandwidth will effectively be saturated. Another key point is that the respective average additional and idle costs when using synchronous and asynchronous were minimal, implying operations outside of directly computing tasks and sending data have little effect on overall system performance. Note that these costs do rise when using four computation threads, though this is likely due to the gRPC threads waiting for processor time before being able to handle outbound data.

Thread #	Total Time (s)	Total Mult Time (s)	Total Add Time (s)	Avg. Comm Cost (s)	Avg. Mult Cost (s)	Avg. Addi Cost (s)
1 Thread	83.382919	81.349977	2.684011	0.020414	0.628687	0.000026
2 Thread	42.175365 (1.98x)	41.340792 (1.97x)	1.492626 (1.80x)	0.022142	0.632777	0.000050
3 Thread	28.149070 (2.96x)	27.365983 (2.97x)	1.439219 (1.86x)	0.024781	0.628498	0.000059
4 Thread	21.748822 (3.83x)	21.707201 (3.75x)	1.374750 (1.95x)	0.024082	0.645349	0.000816

Table 4.1: Table of total times and task costs when performing synchronous multiplication of two 2050x2050 matrices using 410x410 block sizes with one to four computation threads.

Thread #	Total Time (s)	Total Mult Time (s)	Total Add Time (s)	Avg. Comm Cost (s)	Avg. Mult Cost (s)	Avg. Idle Cost (s)
1 Thread	29.979391	29.350825	1.098870	0.011331	0.234638	0.002518
2 Thread	15.194023 (1.97x)	14.758651 (1.99x)	0.905860 (1.21x)	0.014708	0.234498	0.004256
3 Thread	10.241340 (2.93x)	9.880388 (2.97x)	0.835543 (1.32x)	0.017647	0.234537	0.005583
4 Thread	7.974954 (3.76x)	7.803008 (3.76x)	0.858051 (1.28x)	0.019131	0.241335	0.006617

Table 4.2: Table of total times and task costs when performing asynchronous multiplication of two 1475x1475 matrices using 295x295 block sizes with one to four computation threads.

Tables 4.3 and 4.4 display the various data rates from executing at 410x410 and 295x295 submatrix sizes using the synchronous and asynchronous strategies, respectively. In general, the effective data rate at each size corresponded closely with the multiplication data rate. This makes sense as the average communication cost is significantly lower than the average multiplication cost, meaning the bottleneck is the computation time. Additionally, asynchronous was generally found to have a faster effective data rate than its synchronous counterpart when performing multiplication tasks at all submatrix sizes and thread configurations. While some of this benefit is

due to asynchronous' background communication, the benefit is likely primarily due to the use of smaller submatrix sizes. Consider the multiplication data rates in tables 4.3 and 4.4. Asynchronous' multiplication data rate is significantly faster despite this measurement only accounting for matrix multiplication time. This is expected behaviour if we consider that smaller matrix sizes correspond with faster matrix multiplication rates due to fewer cache misses at each cache level (see section 3.1). Additionally, little benefit is offered by asynchronous, as the average communication cost is significantly smaller than the average mult cost. This means the penalty associated with having each computation thread idle for communication in synchronous is effectively negligible. Despite this, further experimentation is required to validate this hypothesis.

Thread #	Mult task #	Mult data rate (MB/s)	Comm data rate (MB/s)	Eff. data rate (MB/s)
1 Thread	125	2.139	98.814	2.066
2 Thread	125	4.250	91.103	4.066
3 Thread	125	6.419	81.410	6.143
4 Thread	125	8.335	83.764	7.744

Table 4.3: Table of the multiplication input data rate, effective communication data rate and effective data rate when performing synchronous multiplication of two 2050x2050 matrices using 410x410 block sizes with one to four computation threads.

Thread #	Mult task #	Mult data rate (MB/s)	Comm data rate (MB/s)	Eff. data rate (MB/s)
1 Thread	125	2.967	92.163	2.965
2 Thread	125	5.938	71.002	5.897
3 Thread	125	8.905	59.177	8.713
4 Thread	125	11.539	54.587	11.153

Table 4.4: Table of the multiplication input data rate, effective communication data rate and effective data rate when performing asynchronous multiplication of two 1475x1475 using 295x295 block sizes with one to four computation threads.

Interestingly, non-linear behaviour multiplication time was observed due to a communication bottleneck when conducting experiments at 70x70 and 52x52 submatrix sizes for the synchronous and asynchronous strategies, respectively (see Table 4.5). Consider the effective communication and multiplication data rates when using four threads in Table 4.6. The communication rate is 32.325 MB/s, while the multiplication rate is 39.132 MB/s. This effectively means computation threads will be consuming input data faster than task input data is passed into the RPI due to a communication

bottleneck. This bottleneck effectively negates the added benefit from the fourth execution thread. Additionally, the effective communication rate is significantly below the theoretical limit of 125 MB/s offered by the Ethernet, highlighting an issue with gRPC when using smaller data sizes. One potential cause for this poor performance is a high overhead cost when sending each message within asynchronous gRPC streaming. This cost could be longer than the actual transmission itself at small data sizes, though further investigation is required to validate this. Assuming this communication problem could be addressed, using small block sizes will likely offer the best effective data rate due to L1 cache's significantly faster multiplication time. We should also note that the measured effective communication data rate worsened as more computation threads were added. This is likely due to a bottleneck within the gRPC implementation on the primary's side when sending more data simultaneously.

Thread #	Total Time (s)	Total Mult Time (s)	Total Add Time (s)	Avg. Comm Cost (s)	Avg. Mult Cost (s)	Avg. Addi Cost (s)
1 Thread	0.838796	0.666351	0.177484	0.001774	0.003328	0.000009
2 Thread	0.427328 (1.96x)	0.345663 (1.93x)	0.086718 (2.05x)	0.001697	0.003577	0.000018
3 Thread	0.286247 (2.93x)	0.241358 (2.76x)	0.050105 (3.54x)	0.001563	0.003764	0.000021
4 Thread	0.267144 (3.13x)	0.224400 (2.97x)	0.048630 (3.64x)	0.001819	0.004007	0.000395

Table 4.5: Table of total times and task costs when performing synchronous multiplication of two 350x350 matrices using 70x70 block sizes with one to four computation threads.

Thread #	Mult task #	Mult data rate (MB/s)	Comm data rate (MB/s)	Eff. data rate (MB/s)
1 Thread	125	11.779	33.145	7.353
2 Thread	125	21.918	34.649	14.176
3 Thread	125	31.243	37.620	20.302
4 Thread	125	39.132	32.325	21.875

Table 4.6: Table of the multiplication input data rate, effective communication data rate and effective data rate when performing synchronous multiplication of two 350x350 matrices using 70x70 block sizes with one to four computation threads.

4.6 Multi device scalability

The objective of this experiment was to determine whether the synchronous and asynchronous systems were scalable as the number of secondary RPIs was varied. As seen in figure 4.3, the experiment was set up by running the secondary application

on a number of RPI 5 devices, which are each connected to a switch using a 1Gb/s Ethernet link. Additionally, the primary was also connected to the switch using a 2.5 Gb/s Ethernet link. Similar to the single device scalability experiment, the various total times were measured and analysed as the number of RPIs used was varied from one to three. Ideally, the total mult time should linearly decrease based on the total number of computation threads, irrespective of the number of RPIs used. Due to this, each RPI configuration was run using one to four computation threads. Additionally, the experiment was run with submatrix block sizes corresponding to the L3 cache sizes (i.e. 410x410 for synchronous and 295x295 for asynchronous) as it offered good single device scalability with little to no communication bottleneck. This section will offer an overview of key aspects from the experiments, the full set of results is available in Appendix B.

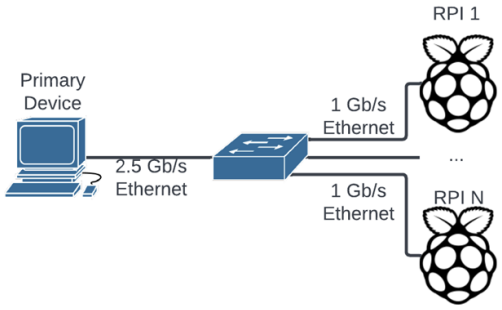

Figure 4.3: Diagram of multi device experiment. The number of RPIs connect to the switch varies between one and three.

figure 4.4 shows the total mult time decrease as the number of computation threads was varied when running experiments with one to three RPIs. In general, the number of computation threads almost linearly decreases total multiplication time until it reaches eight threads, where the benefit of additional threads tapers off.

This drop off is likely due to a communication bottleneck forming with additional computation threads. The theoretical inbound data rate on the RPI is 125 MB/s (see section 3.1), meaning the communication rate for each RPI should be near this limit, however, this is clearly not always the case as the number of computation threads was increased during the multi device experiment. Consider the data rates when performing synchronous and asynchronous multiplication from using three RPIs with one to four computation threads in figure 4.5. The effective data rate generally aligns with the multiplication data rate. Despite this, the communication rate appears to drop with additional computation threads. The communication data rate eventually drops below

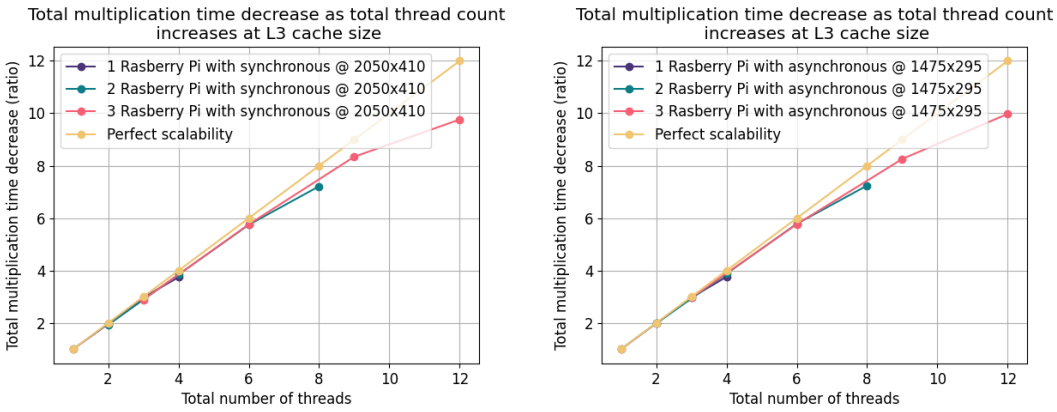

Figure 4.4: Plots of the total multiplication time decrease as total computation thread count varies when using synchronous and asynchronous operation. Different coloured lines correspond when different number of total RPIs are used. Note that the yellow line corresponds to perfect linearity at each computation thread count.

the multiplication data rate, causing a communication bottleneck (i.e. the effective data rate no longer aligns with the multiplication data rate). Two potential causes for this bottle are: 1) the primary is utilising its full Ethernet bandwidth and is hence unable to transmit data to RPI at the theoretical rate, and 2) the primary's implementation has a bottleneck that is affecting the communication data rate.

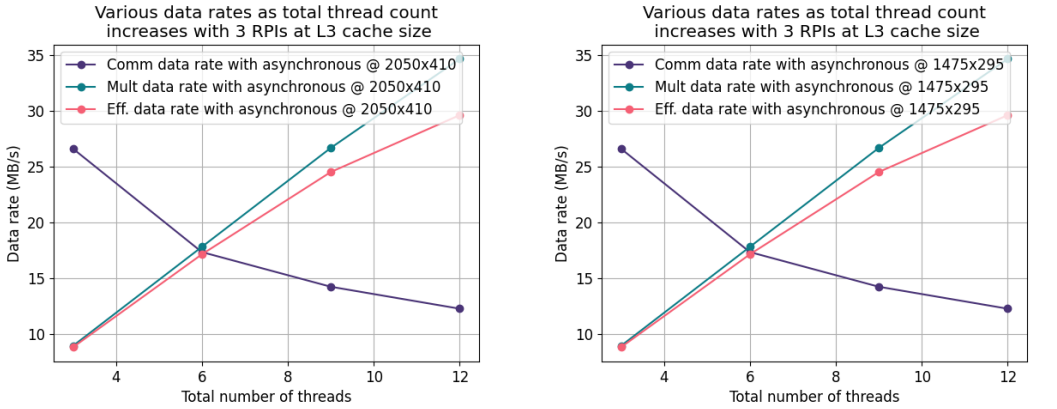

Figure 4.5: Plots of the data rates as total computation thread count varies when using synchronous and asynchronous operation on three RPIs.

To test the first hypothesis for the low communication data rate, the experiment was run again using three RPIs each with four computation threads, with the outbound data rate sent from the primary to all secondaries being measured using Wireshark [43]. The outbound data rate when performing multiplications was measured to be $\sim 37MB/s$, which corresponds almost exactly to the total bandwidth required to feed three RPIs at a

rate of 12.235 MB/s (i.e. $3 * 12.235\ MB/s \sim 37 MB/s$), meaning the primary's Ethernet is not the bottleneck.

The bottleneck is likely caused by the primary's implementation failing to scale and not the secondary. As discussed in section 4.5, minimal change was observed in the additional and idle times when increasing the number of threads. Given that each secondary operates independently from each other due to the primary-secondary communication model, these values should remain the same, assuming the primary scales ideally for each additional device. As seen in figure 4.6, this is not the case as the additional cost and idle cost both increase when more RPIs are used during the multi device scalability experiment. This implies the primary itself is failing to scale. Further investigation is required to determine which aspect of the primary is failing the scale. Two potential culprits include the gRPC's internal library and the software architecture used by the primary.

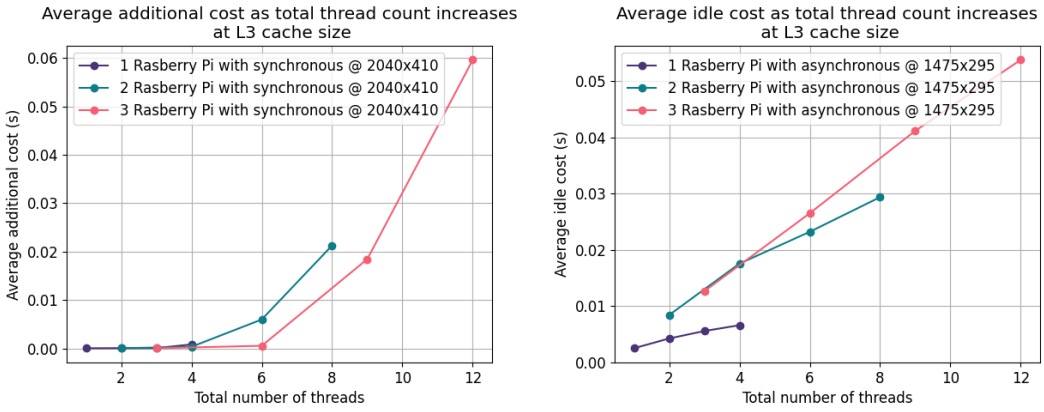

Figure 4.6: Plots of additional and idle cost for synchronous and asynchronous respectively as total computation thread count varies. Different coloured lines correspond when different number of total RPIs are used.

Chapter 5

Conclusion

This chapter evaluates the project's success based on the objectives set in Chapter 1 and summarises the results gathered from the various experiments conducted in Chapter 4. Additionally, potential future project work has been evaluated and considered.

5.1 Key findings

This project aimed to develop a suitable framework for distributing large matrix multiplication across numerous RPIs. This was achieved through developing a task-based primary-secondary framework which successfully performs distributed matrix multiplication across multiple RPIs using the synchronous and asynchronous modes of operation via Ethernet using gRPC.

Two experiments were conducted to evaluate the implemented system's scalability on a single and multiple RPIs. Each of these experiments analysed system performance based on the total time from the primary's perspective, the costs associated with each task from the secondary's perspective, and the rate at which data passed from the primary to the secondary is consumed. It was found that both synchronous and asynchronous generally offered near-linear scalability with each additional core-pinned computation thread on a single device. Diminishing returns was observed when using four computation threads due to time sharing between a computation thread and gRPC communication. While a communication bottleneck was observed when using L1 cache-sized buffers, the system generally had a higher effective data consumption rate when smaller submatrix block sizes were used. Additionally, both synchronous and asynchronous had near-linear scalability when using additional RPIs, with a performance drop off as the total number of computation threads increased. The cause

of this dropoff was identified to be likely casued by a bottleneck in how the primary on-loads and off-loads tasks to the secondaries. If challenges in the primary were addressed, near-linear scalability is probable as the number of RPIs and computation threads increases.

5.2 Limitations and future work

Both the single-device and multi-device scalability experiments have key limitations that must be considered. We assumed that there exists a primary with infinite computation and communication capabilities. We attempted to demonstrate these using a simplistic primary-secondary-based distributed system that assumes the primary has infinite computation and communication capabilities. In our experiments, our primary was limited by it's hardware and the primary's code implementation. These likely affected the communication data rate as the number of computation threads increased, causing a bottleneck. Due to this, this current implementation *is not* suitable for large scalability across hundreds of RPIs, which would likely be required to offer comparable to a state-of-the-art HPC cluster.

Future work tends to fall into two categories: 1) improving the current primary-secondary-based system, and 2) expanding work to use more sophisticated architectures. In terms of system improvements, the primary's implementation was identified as the likely culprit in diminishing scalability with additional RPIs. This could be investigated and addressed to develop a more optimised system. Additionally, our task allocation system is very simplistic, as we treat each RPI core as a separate computation unit. Investigations could be done to see if utilising all RPI cores to execute a single task using a parallel matrix multiplication algorithm is a better approach. Additionally, each task was independent and required both input matrices to be transmitted for each task. Given that contemporary distributed matrix multiplication systems utilise local storage to optimise communication time, perhaps the primary and secondaries can take advantage of this through intelligent task allocation. Additionally, little literature exists on the scalability of contemporary distributed matrix multiplication algorithms on edge devices. Future work could integrate these systems into the existing framework and conduct experiments to determine their viability on RPI 5s. Given that many contemporary systems rely on all-to-all communication, they will likely struggle with scalability when using hundreds of RPIs. Work could then be done to determine whether a tree-based distributed topology aid in achieving scalability using hundreds of RPIs.

Bibliography

[1] Najmul Hassan, Saira Gillani, Ejaz Ahmed, Ibrar Yaqoob, and Muhammad Imran. The role of edge computing in internet of things. *IEEE Communications Magazine*, 56(11):110–115, 2018.

[2] M. G. Sarwar Murshed, Christopher Murphy, Daqing Hou, Nazar Khan, Ganesh Ananthanarayanan, and Faraz Hussain. Machine learning at the network edge: A survey. *ACM Comput. Surv.*, 54(8), oct 2021.

[3] Lionel Sujay Vailshery. Iot connected devices worldwide 2019-2030, Jul 2023.

[4] Mohammad Loni, Sima Sinaei, Ali Zoljodi, Masoud Daneshtalab, and Mikael Sjödin. Deepmaker: A multi-objective optimization framework for deep neural networks in embedded systems. *Microprocessors and Microsystems*, 73:102989, 2020.

[5] Ji Lin, Ligeng Zhu, Wei-Ming Chen, Wei-Chen Wang, Chuang Gan, and Song Han. On-device training under 256kb memory, 2022.

[6] Johannes de Fine Licht, Grzegorz Kwasniewski, and Torsten Hoefler. Flexible communication avoiding matrix multiplication on fpga with high-level synthesis. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '20, page 244–254, New York, NY, USA, 2020. Association for Computing Machinery.

[7] Stephen Cass. Taking ai to the edge: Google's tpu now comes in a maker-friendly package. *IEEE Spectrum*, 56(5):16–17, 2019.

[8] Longguang Wang, Xiaoyu Dong, Yingqian Wang, Xinyi Ying, Zaiping Lin, Wei An, and Yulan Guo. Exploring sparsity in image super-resolution for efficient inference. In *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4915–4924, 2021.

[9] Chih-Chia Lin, Chia-Yin Liu, Chih-Hsuan Yen, Tei-Wei Kuo, and Pi-Cheng Hsiu. Intermittent-aware neural network pruning. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2023.

[10] Ashkan Yousefpour, Genya Ishigaki, Riti Gour, and Jason P. Jue. On reducing iot service delay via fog offloading. *IEEE Internet of Things Journal*, 5(2):998–1010, 2018.

[11] He Li, Kaoru Ota, and Mianxiong Dong. Learning iot in edge: Deep learning for the internet of things with edge computing. *IEEE Network*, 32(1):96–101, 2018.

[12] Meng Wang, Liang Qian, Na Meng, Yusong Cheng, and Weiwei Fang. Model parallelism optimization for distributed dnn inference on edge devices. In *2023 IEEE 14th International Symposium on Parallel Architectures, Algorithms and Programming (PAAP)*, pages 1–6, 2023.

[13] Weiwei Miao, Zeng Zeng, Lei Wei, Shihao Li, Chengling Jiang, and Zhen Zhang. Adaptive dnn partition in edge computing environments. In *2020 IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 685–690, 2020.

[14] Dask core developers. Dask. `https://www.dask.org/`, 2022. [Accessed 10-08-2024].

[15] Apache Software Foundation. Apache spark. `https://spark.apache.org/`, 2018. [Accessed 10-08-2024].

[16] Siddharth Bhave, Matt Tolentino, Henry Zhu, and Jie Sheng. Embedded middleware for distributed raspberry pi device to enable big data applications. In *2017 IEEE International Conference on Computational Science and Engineering (CSE) and IEEE International Conference on Embedded and Ubiquitous Computing (EUC)*, volume 2, pages 103–108, 2017.

[17] Jose Paolo Talusan, Francis Tiausas, Sopicha Stirapongsasuti, Yugo Nakamura, Teruhiro Mizumoto, and Keiichi Yasumoto. Evaluating performance of in-situ distributed processing on iot devices by developing a workspace context recognition service. In *2019 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, pages 633–638, 2019.

[18] G.H. Golub and C.F. Van Loan. *Matrix Computations*. Johns Hopkins Studies in the Mathematical Sciences. Johns Hopkins University Press, 2013.

[19] Khaled Thabet and Sumaia Al-Ghuribi. Matrix multiplication algorithms. *International Journal of Computer Science and Network Security (IJCSNS)*, 12(2):74, 2012.

[20] Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, August 1969.

[21] Joost Verbraeken, Matthijs Wolting, Jonathan Katzy, Jeroen Kloppenburg, Tim Verbelen, and Jan S. Rellermeyer. A survey on distributed machine learning. *ACM Comput. Surv.*, 53(2), mar 2020.

[22] Eric P. Xing, Qirong Ho, Pengtao Xie, and Dai Wei. Strategies and principles of distributed machine learning on big data. *Engineering*, 2(2):179–195, 2016.

[23] T. Cheatham, A. Fahmy, D.C. Stefanescu, and L.G. Valiant. Bulk synchronous parallel computing-a paradigm for transportable software. In *Proceedings of the Twenty-Eighth Annual Hawaii International Conference on System Sciences*, volume 2, pages 268–275 vol.2, 1995.

[24] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, jan 2008.

[25] Nenad Anchev, Marjan Gusev, Sasko Ristov, and Blagoj Atanasovski. Some optimization techniques of the matrix multiplication algorithm. In *Proceedings of the ITI 2013 35th International Conference on Information Technology Interfaces*, pages 71–76, 2013.

[26] Xia Liao, Shengguo Li, Wei Yu, and Yutong Lu. Parallel matrix multiplication algorithms in supercomputing. In *2021 6th International Conference on Intelligent Computing and Signal Processing (ICSP)*, pages 1–4, 2021.

[27] Nikola Tomikj and Marjan Gusev. Parallel matrix multiplication. In *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 0204–0209, 2018.

[28] Lynn Elliot Cannon. *A cellular computer to implement the kalman filter algorithm*. PhD thesis, USA, 1969. AAI7010025.

[29] Jaeyoung Choi, Jack J. Dongarra, and David W. Walker. Pumma: Parallel universal matrix multiplication algorithms on distributed memory concurrent computers. *Concurrency: Practice and Experience*, 6(7):543–570, October 1994.

[30] R. A. Van De Geijn and J. Watts. Summa: scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience*, 9(4):255–274, April 1997.

[31] Edgar Solomonik and James Demmel. Communication-optimal parallel 2.5d matrix multiplication and lu factorization algorithms. In Emmanuel Jeannot, Raymond Namyst, and Jean Roman, editors, *Euro-Par 2011 Parallel Processing*, pages 90–109, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[32] S Huss-Lederman, E M Jacobson, A Tsao, and G Zhang. Matrix multiplication on the intel touchstone delta. dec 1993.

[33] Grey Ballard, James Demmel, Olga Holtz, Benjamin Lipshitz, and Oded Schwartz. Communication-optimal parallel algorithm for strassen's matrix multiplication, 2012.

[34] James Demmel, David Eliahu, Armando Fox, Shoaib Kamil, Benjamin Lipshitz, Oded Schwartz, and Omer Spillinger. Communication-optimal parallel recursive rectangular matrix multiplication. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 261–272, 2013.

[35] Grzegorz Kwasniewski, Marko Kabić, Maciej Besta, Joost VandeVondele, Raffaele Solcà, and Torsten Hoefler. Red-blue pebbling revisited: near optimal parallel matrix-matrix multiplication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, New York, NY, USA, 2019. Association for Computing Machinery.

[36] Raspberry Pi Foundation. Raspberry pi 5 product brief. 2024.

[37] Arm Holdings plc. Arm cortex-a76 core technical reference manual. pages 76–100, 2018.

[38] Arm Holdings plc. Arm® cortex®-a76 software optimization guide. pages 9–12, 2019.

[39] Arm Holdings plc. Armv8-a instruction set architecture. pages 11–14, 2020.

[40] Mpich. `https://www.mpich.org/`. [Accessed 20-04-2024].

[41] gRPC Authors. Core concepts, architecture and lifecycle. https://grpc.io/docs/what-is-grpc/core-concepts/, Jul 2024. [Accessed 05-08-2024].

[42] Nick Clifford, Andries Brouwer, and Michael Kerrisk. clock_gettime(3) — linux manual page. https://www.man7.org/linux/man-pages/man3/clock_gettime.3.html, Apr 2020. [Accessed 15-08-2024].

[43] Wireshark Foundation. Wireshark. https://www.wireshark.org/. [Accessed 21-08-2024].

Appendix A

Single device scalability results

A.1 Synchronous results

Thread #	Total Time (s)	Total Mult Time (s)	Total Add Time (s)	Avg. Comm Cost (s)	Avg. Mult Cost (s)	Avg. Addi Cost (s)
1 Thread	0.838796	0.666351	0.177484	0.001774	0.003328	0.000009
2 Thread	0.427328 (1.96x)	0.345663 (1.93x)	0.086718 (2.05x)	0.001697	0.003577	0.000018
3 Thread	0.286247 (2.93x)	0.241358 (2.76x)	0.050105 (3.54x)	0.001563	0.003764	0.000021
4 Thread	0.267144 (3.13x)	0.224400 (2.97x)	0.048630 (3.64x)	0.001819	0.004007	0.000395

Table A.1: Table of total times and task costs when performing synchronous multiplication of two 350x350 matrices using 70x70 block sizes with one to four computation threads.

Thread #	Mult task #	Mult data rate (MB/s)	Comm data rate (MB/s)	Eff. data rate (MB/s)
1 Thread	125	11.779	33.145	7.353
2 Thread	125	21.918	34.649	14.176
3 Thread	125	31.243	37.620	20.302
4 Thread	125	39.132	32.325	21.875

Table A.2: Table of the multiplication input data rate, effective communication data rate and effective data rate when performing synchronous multiplication of two 350x350 matrices using 70x70 block sizes with one to four computation threads.

Thread #	Total Time (s)	Total Mult Time (s)	Total Add Time (s)	Avg. Comm Cost (s)	Avg. Mult Cost (s)	Avg. Addi Cost (s)
1 Thread	11.639586	11.093248	0.634177	0.005664	0.082346	0.000015
2 Thread	5.914058 (1.97x)	5.658569 (1.96x)	0.342962 (1.85x)	0.006255	0.082741	0.000035
3 Thread	4.005656 (2.91x)	3.761507 (2.95x)	0.332185 (1.91x)	0.007032	0.082665	0.000054
4 Thread	3.165573 (3.86x)	2.952456 (3.76x)	0.324228 (1.96x)	0.008482	0.084373	0.000306

Table A.3: Table of total times and task costs when performing synchronous multiplication of two 1040x1040 matrices using 208x208 block sizes with one to four computation threads.

Thread #	Mult task #	Mult data rate (MB/s)	Comm data rate (MB/s)	Eff. data rate (MB/s)
1 Thread	125	4.203	91.661	3.900
2 Thread	125	8.366	83.001	7.646
3 Thread	125	12.561	73.829	11.502
4 Thread	125	16.409	61.208	14.654

Table A.4: Table of the multiplication input data rate, effective communication data rate and effective data rate when performing synchronous multiplication of two 2040x2040 matrices using 208x208 block sizes with one to four computation threads.

Thread #	Total Time (s)	Total Mult Time (s)	Total Add Time (s)	Avg. Comm Cost (s)	Avg. Mult Cost (s)	Avg. Addi Cost (s)
1 Thread	83.382919	81.349977	2.684011	0.020414	0.628687	0.000026
2 Thread	42.175365 (1.98x)	41.340792 (1.97x)	1.492626 (1.80x)	0.022142	0.632777	0.000050
3 Thread	28.149070 (2.96x)	27.365983 (2.97x)	1.439219 (1.86x)	0.024781	0.628498	0.000059
4 Thread	21.748822 (3.83x)	21.707201 (3.75x)	1.374750 (1.95x)	0.024082	0.645349	0.000816

Table A.5: Table of total times and task costs when performing synchronous multiplication of two 2050x2050 matrices using 410x410 block sizes with one to four computation threads.

Thread #	Mult task #	Mult data rate (MB/s)	Comm data rate (MB/s)	Eff. data rate (MB/s)
1 Thread	125	2.139	98.814	2.066
2 Thread	125	4.250	91.103	4.066
3 Thread	125	6.419	81.410	6.143
4 Thread	125	8.335	83.764	7.744

Table A.6: Table of the multiplication input data rate, effective communication data rate and effective data rate when performing synchronous multiplication of two 2050x2050 matrices using 410x410 block sizes with one to four computation threads.

A.2 Asynchronous results

Thread #	Total Time (s)	Total Mult Time (s)	Total Add Time (s)	Avg. Comm Cost (s)	Avg. Mult Cost (s)	Avg. Idle Cost (s)
1 Thread	0.279061	0.205391	0.076680	0.001614	0.001350	0.000442
2 Thread	0.135048 (2.07x)	0.101609 (2.02x)	0.036305 (2.11x)	0.001324	0.001427	0.000332
3 Thread	0.109998 (2.54x)	0.073394 (2.79x)	0.039690 (1.93x)	0.001430	0.001395	0.000540
4 Thread	0.106473 (2.62x)	0.072277 (2.84x)	0.038353 (2.48x)	0.001701	0.001496	0.000864

Table A.7: Table of total times and task costs when performing asynchronous multiplication of two 260x260 matrices using 52x52 block sizes with one to four computation threads.

Thread #	Mult task #	Mult data rate (MB/s)	Comm data rate (MB/s)	Eff. data rate (MB/s)
1 Thread	125	16.024	20.104	13.165
2 Thread	125	30.318	44.411	26.612
3 Thread	125	46.520	41.119	36.842
4 Thread	125	57.840	19.087	37.411

Table A.8: Table of the multiplication input data rate, effective communication data rate and effective data rate when performing asynchronous multiplication of two 260x260 using 52x52 block sizes with one to four computation threads.

Thread #	Total Time (s)	Total Mult Time (s)	Total Add Time (s)	Avg. Comm Cost (s)	Avg. Mult Cost (s)	Avg. Idle Cost (s)
1 Thread	3.837359	3.675685	0.216154	0.003264	0.029249	0.000636
2 Thread	1.977329 (1.94x)	1.847551 (1.99x)	0.187388 (1.15x)	0.004416	0.029218	0.001161
3 Thread	1.355275 (2.83x)	1.234331 (2.98x)	0.179253 (1.19x)	0.005409	0.029044	0.001658
4 Thread	1.098109 (3.59x)	0.997508 (3.68x)	0.180410 (1.20x)	0.006955	0.030444	0.002178

Table A.9: Table of total times and task costs when performing asynchronous multiplication of two 735x735 matrices using 147x147 block sizes with one to four computation threads.

Thread #	Mult task #	Mult data rate (MB/s)	Comm data rate (MB/s)	Eff. data rate (MB/s)
1 Thread	125	5.910	79.445	5.879
2 Thread	125	11.833	58.720	11.696
3 Thread	125	17.856	47.940	17.507
4 Thread	125	22.713	37.284	21.663

Table A.10: Table of the multiplication input data rate, effective communication data rate and effective data rate when performing asynchronous multiplication of two 735x735 using 147x147 block sizes with one to four computation threads.

Thread #	Total Time (s)	Total Mult Time (s)	Total Add Time (s)	Avg. Comm Cost (s)	Avg. Mult Cost (s)	Avg. Idle Cost (s)
1 Thread	29.979391	29.350825	1.098870	0.011331	0.234638	0.002518
2 Thread	15.194023 (1.97x)	14.758651 (1.99x)	0.905860 (1.21x)	0.014708	0.234498	0.004256
3 Thread	10.241340 (2.93x)	9.880388 (2.97x)	0.835543 (1.32x)	0.017647	0.234537	0.005583
4 Thread	7.974954 (3.76x)	7.803008 (3.76x)	0.858051 (1.28x)	0.019131	0.241335	0.006617

Table A.11: Table of total times and task costs when performing asynchronous multiplication of two 1475x1475 matrices using 295x295 block sizes with one to four computation threads.

Thread #	Mult task #	Mult data rate (MB/s)	Comm data rate (MB/s)	Eff. data rate (MB/s)
1 Thread	125	2.967	92.163	2.965
2 Thread	125	5.938	71.002	5.897
3 Thread	125	8.905	59.177	8.713
4 Thread	125	11.539	54.587	11.153

Table A.12: Table of the multiplication input data rate, effective communication data rate and effective data rate when performing asynchronous multiplication of two 1475x1475 using 295x295 block sizes with one to four computation threads.

Appendix B

Multi device scalability

B.1 Synchronous results

RPI #	Total Time (s)	Total Mult Time (s)	Total Add Time (s)	Avg. Comm Cost (s)	Avg. Mult Cost (s)	Avg. Addi Cost (s)
1 RPI	83.382919	81.349977	2.684011	0.020414	0.628687	0.000026
2 RPI	44.177361 (1.89x)	42.132450 (1.93x)	2.731622 (0.98x)	0.043205	0.627307	0.000031
3 RPI	30.176473 (2.76x)	28.129135 (2.89x)	2.826389 (0.95x)	0.052589	0.627960	0.000030

Table B.1: Table of total times and task costs when performing synchronous multiplication of two 2050x2050 matrices using 410x410 block sizes with one computation thread on one to three RPIs.

RPI #	Mult task #	Mult data rate (MB/s)	Comm data rate (MB/s)	Eff. data rate (MB/s)
1 RPI	125	2.068	98.815	2.066
2 RPI	125	4.024	46.689	3.990
3 RPI	125	6.068	38.358	5.976

Table B.2: Table of the multiplication input data rate, effective communication data rate and effective data rate when performing asynchronous multiplication of two 2050x2050 using 410x410 block sizes with one computation thread on one to three RPIs.

RPI #	Total Time (s)	Total Mult Time (s)	Total Add Time (s)	Avg. Comm Cost (s)	Avg. Mult Cost (s)	Avg. Addi Cost (s)
1 RPI	42.175365	41.340792	1.492626	0.022142	0.632777	0.000050
2 RPI	22.919571 (1.84x)	21.146497 (1.95x)	2.481691 (0.60x)	0.056741	0.627696	0.000328
3 RPI	16.080611 (2.62x)	14.115513 (2.93x)	2.720289 (0.55x)	0.077638	0.627978	0.000537

Table B.3: Table of total times and task costs when performing synchronous multiplication of two 2050x2050 matrices using 410x410 block sizes with two computation threads on one to three RPIs.

RPI #	Mult task #	Mult data rate (MB/s)	Comm data rate (MB/s)	Eff. data rate (MB/s)
1 RPI	125	4.250	91.103	4.066
2 RPI	125	8.570	35.551	7.949
3 RPI	125	12.849	25.982	11.909

Table B.4: Table of the multiplication input data rate, effective communication data rate and effective data rate when performing asynchronous multiplication of two 2050x2050 using 410x410 block sizes with two computation threads on one to three RPIs.

RPI #	Total Time (s)	Total Mult Time (s)	Total Add Time (s)	Avg. Comm Cost (s)	Avg. Mult Cost (s)	Avg. Addi Cost (s)
1 RPI	28.149070	27.365983	1.439219	0.024781	0.628498	0.000059
2 RPI	15.830197 (1.78x)	14.153660 (1.93x)	2.426517 (0.59x)	0.066770	0.625634	0.005974
3 RPI	11.753083 (2.40x)	9.750247 (2.81x)	2.835532 (0.51x)	0.098655	0.627042	0.018444

Table B.5: Table of total times and task costs when performing synchronous multiplication of two 2050x2050 matrices using 410x410 block sizes with three computation threads on one to three RPIs.

RPI #	Mult task #	Mult data rate (MB/s)	Comm data rate (MB/s)	Eff. data rate (MB/s)
1 RPI	125	6.202	81.401	6.143
2 RPI	125	12.073	30.211	11.877
3 RPI	125	17.917	20.447	17.241

Table B.6: Table of the multiplication input data rate, effective communication data rate and effective data rate when performing asynchronous multiplication of two 2050x2050 using 410x410 block sizes with three computation threads on one to three RPIs.

RPI #	Total Time (s)	Total Mult Time (s)	Total Add Time (s)	Avg. Comm Cost (s)	Avg. Mult Cost (s)	Avg. Addi Cost (s)
1 RPI	21.748822	21.707201	1.374750	0.024082	0.645349	0.000816
2 RPI	12.859698 (1.69x)	11.273037 (1.92x)	2.445517 (0.56x)	0.076921	0.639830	0.021274
3 RPI	10.224802 (2.13x)	8.332107 (2.61x)	2.983663 (0.46x)	0.122284	0.638648	0.059648

Table B.7: Table of total times and task costs when performing synchronous multiplication of two 2050x2050 matrices using 410x410 block sizes with four computation threads on one to three RPIs.

RPI #	Mult task #	Mult data rate (MB/s)	Comm data rate (MB/s)	Eff. data rate (MB/s)
1 RPI	125	8.335	83.764	7.744
2 RPI	125	16.814	26.224	14.911
3 RPI	125	25.268	16.496	20.145

Table B.8: Table of the multiplication input data rate, effective communication data rate and effective data rate when performing asynchronous multiplication of two 2050x2050 using 410x410 block sizes with four computation threads on one to three RPIs.

B.2 Asynchronous results

RPI #	Total Time (s)	Total Mult Time (s)	Total Add Time (s)	Avg. Comm Cost (s)	Avg. Mult Cost (s)	Avg. Idle Cost (s)
1 RPI	29.979391	29.350825	1.098870	0.011331	0.234638	0.002518
2 RPI	15.669458 (1.90x)	14.810790 (1.97x)	1.335979 (0.81x)	0.014925	0.2340858	0.008431
3 RPI	10.826506 (2.77x)	9.910222 (2.96x)	1.404478 (0.78x)	0.039292	0.234594	0.012655

Table B.9: Table of total times and task costs when performing asynchronous multiplication of two 1475x1475 matrices using 295x295 block sizes with one computation thread on one to three RPIs.

RPI #	Mult task #	Mult data rate (MB/s)	Comm data rate (MB/s)	Eff. data rate (MB/s)
1 RPI	125	2.967	92.163	2.965
2 RPI	125	5.948	69.970	5.876
3 RPI	125	8.903	26.578	8.781

Table B.10: Table of the multiplication input data rate, effective communication data rate and effective data rate when performing asynchronous multiplication of two 1475x1475 matrices using 295x295 block sizes with one computation thread on one to three RPIs.

RPI #	Total Time (s)	Total Mult Time (s)	Total Add Time (s)	Avg. Comm Cost (s)	Avg. Mult Cost (s)	Avg. Idle Cost (s)
1 RPI	15.194023	14.758651	0.905860	0.014708	0.234498	0.004256
2 RPI	8.442334 (1.80x)	7.563222 (1.95x)	1.381860 (0.66x)	0.043767	0.234575	0.017574
3 RPI	6.08550 (2.50x)	5.085923 (2.90x)	1.542304 (0.59x)	0.060326	0.234942	0.026541

Table B.11: Table of total times and task costs when performing asynchronous multiplication of two 1475x1475 matrices using 295x295 block sizes with two computation threads on one to three RPIs.

RPI #	Mult task #	Mult data rate (MB/s)	Comm data rate (MB/s)	Eff. data rate (MB/s)
1 RPI	125	5.938	71.002	5.897
2 RPI	125	11.872	23.860	11.506
3 RPI	125	17.780	17.311	17.111

Table B.12: Table of the multiplication input data rate, effective communication data rate and effective data rate when performing asynchronous multiplication of two1475x1475 matrices using 295x295 block sizes with two computation threads on one to three RPIs.

RPI #	Total Time (s)	Total Mult Time (s)	Total Add Time (s)	Avg. Comm Cost (s)	Avg. Mult Cost (s)	Avg. Idle Cost (s)
1 RPI	10.241340	9.880388	0.835543	0.017647	0.234537	0.005583
2 RPI	5.935398 (1.72x)	5.073384 (1.94x)	1.398888 (0.60x)	0.048160	0.234586	0.023237
3 RPI	4.597594 (2.22x)	3.553982 (2.78x)	1.638006 (0.51x)	0.073521	0.235124	0.041199

Table B.13: Table of total times and task costs when performing asynchronous multiplication of two 1475x1475 matrices using 295x295 block sizes with three computation threads on one to three RPIs.

RPI #	Mult task #	Mult data rate (MB/s)	Comm data rate (MB/s)	Eff. data rate (MB/s)
1 RPI	125	8.905	59.177	8.808
2 RPI	125	17.807	21.684	17.153
3 RPI	125	26.649	14.204	24.487

Table B.14: Table of the multiplication input data rate, effective communication data rate and effective data rate when performing asynchronous multiplication of two1475x1475 matrices using 295x295 block sizes with three computation threads on one to three RPIs.

RPI #	Total Time (s)	Total Mult Time (s)	Total Add Time (s)	Avg. Comm Cost (s)	Avg. Mult Cost (s)	Avg. Idle Cost (s)
1 RPI	7.974954	7.803008	0.858051	0.019131	0.241335	0.006617
2 RPI	4.808485 (1.66x)	4.051877 (1.93x)	1.408140 (0.61x)	0.049816	0.240558	0.029400
3 RPI	3.904607 (2.04x)	2.941590 (2.65x)	1.661148 (0.51x)	0.0853561	0.241267	0.053823

Table B.15: Table of total times and task costs when performing asynchronous multiplication of two 1475x1475 matrices using 295x295 block sizes with four computation threads on one to three RPIs.

RPI #	Mult task #	Mult data rate (MB/s)	Comm data rate (MB/s)	Eff. data rate (MB/s)
1 RPI	125	11.539	54.587	11.153
2 RPI	125	23.153	20.963	21.478
3 RPI	125	34.627	12.235	29.584

Table B.16: Table of the multiplication input data rate, effective communication data rate and effective data rate when performing asynchronous multiplication of two 1475x1475 matrices using 295x295 block sizes with four computation threads on one to three RPIs.