공학석사 학위논문

A Black-Box Graph Partitioner for Generalized Deep Neural Networks Parallelization

심층 신경망 병렬화를 위한 블랙박스 그래프 분할기

2023년 2 월

서울대학교 대학원

컴퓨터공학부

Jaume Mateu Cuadrat

A Black-Box Graph Partitioner for Generalized Deep Neural Networks Parallelization

지도교수 Bernhard Egger

이 논문을 공학석사 학위논문으로 제출함

2022년 12월

서울대학교 대학원

컴퓨터공학부

Jaume Mateu Cuadrat

Jaume Mateu Cuadrat의 석사 학위논문을 인준함

2022년 12월

위 원 장	이재진	(인)
부위원장	Bernhard Egger	(인)
위 원	김진수	(인)

Abstract

Deep neural networks (DNNs) compromising larger and larger models are being adopted in many domains. Businesses and individuals looking to create deep learning applications require purchasing expensive hardware setups or renting high-end machines from cloud providers, both of which are significant to the customers. An exciting alternative to avoid the high cost of machines powerful enough to run DNNs with billions or trillions of parameters is to use a cheaper but slower set of machines and distribute the workload. Several parallelization strategies have been proposed to tailor the storage and computational requirements to each available device while meeting the application's latency requirements. However, using such setups requires customers to have intricate knowledge of the algorithm or model to devise an efficient plan of workload parallelization. In this thesis, I propose BBGraP, a black-box graph partitioner that is device- and model-agnostic and produces efficient parallelization plans for deep learning inference. The proposed method takes different types of networks as input and generates a workload division that satisfies each node's memory and computational constraints. A graph optimizer eliminates redundant operations, data transfers, and synchronization points to reduce the amount of data transferred while improving a workload's latency. Then an automatic search finds the best partition possible according to the configuration given. As a proof-of-concept, I apply BBGraP to a cluster of distributed nodes and a multicore FPGA. The evaluation shows a speedup up to 2-fold.

Keywords: Deep Neural Networks, Parallelization, Compiler, Automatic Searcher, Black-Box, Graph Optimizer

Student Number: 2020-27033

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Deep neural networks (DNNs) are now ubiquitously adopted to various applications [1,3,11,20]. DNNs consist of numerous hidden layers with weights and massive input data sets combined with DNN parameters not trivial in size, thus requiring a certain amount of memory during inference. Additionally, most layers are matrix multiplications of their weight and the preceding layer's activation, thus making the whole process computationally intensive. Such memory- and compute-intensive DNNs aim to predict with the highest accuracy possible. Furthermore, with bigger DNNs asking for higher prediction accuracy, the number of parameters has now reached trillions [5] for better performances in a trade-off for much large memory and higher computing requirements.

Meanwhile, due to the benefits of flexibility and efficiency, many users and organizations use cloud computing to either use or service various applications [4], including applications with DNNs. The growing popularity of resource-intensive applications now demands high resource usage to accommodate their workloads, but the cost of setting up such high-end node [2] is significant to the customers. One alternative to using a high-end server is to set up a cluster of slower but cheaper ma-

chines to run resource-intensive DNN applications. This alternative requires devising a plan for distributed inference. Distributing or parallelizing inference is one way to decrease total computation and memory requirements. There are a few ways of parallelizing the network, such as data parallelism [10], model parallelism [14], and intra-layer parallelism [17]. Since these methods divide the inference scheme into multiple nodes, they have pros and cons regarding memory and latency trade-offs. Also, the search space for the best parallelization plan can significantly increase by the number of nodes, the number of devices, and the network's size [15]. Another difficulty in hardware configurations and DNN architecture must be considered when devising an efficient parallelization plan to configure the best possible way of dividing the graph into multiple partitions with efforts to reduce memory accesses and fully utilize all devices at hand. Most existing solutions split the DNN into two parts, one running locally or at the edge and the other in the cloud. However, there should be a device-agnostic and model-agnostic planner considering all hardware configurations and DNN layers.

With such problems for parallelization at hand, we propose our framework BB-GraP that can:

1. distribute a graph through different nodes and fully utilizes the devices inside the system.

2. produce a fair distribution among all the nodes without the knowledge of the hardware parameters.

3. optimize through operator pruning and data transfer reduction, reducing the memory footprint inside each mode and the data transfer between different nodes.

4. search the best partition independently from the type of devices and the hetero-

geneity of the devices.

The remaining sections are as follows: Sec. 2 discusses the background and other related works, Sec. 3 explains the process of BBGraP, Sec. 4.4.1 shows how the automatic search of the best workload distribution is done. Sec. 4 shows our evaluation of BBGraP, and Sec. 5 finally concludes this paper.

Chapter 2

Background and Related Works

The parallelism schemes of distributed deep learning are composed of data parallelism, model parallelism, and intra-layer parallelism. Data parallelism distributes input data set to all workers for faster inference with the price of having the full model copied to all workers and thus increasing memory requirements. Model parallelism distributes the model to the workers with possible bottleneck of data transfer between the worker nodes. Intra-layer parallelism divides operators between the workers which in turn requires synchronization to maintain operator semantics.

Different types of data partition and parallelism distributed across devices have been explored in the past. MoDNN [12] divides the network and sends the data via Wi-Fi to different devices. Even though this model can do partition and distribution, the model does this layer-by-layer which requires synchronization after each layer. Another problem is the restriction on the partitions since it's only able to do partition across one layer and only really simple networks which usually are used only in constrained devices.

DeepThings [21] focuses on early layers of the network where the activations are bigger than weights and divide those height- and width-wise. However, DeepThings

lacks flexibility when it comes to the shapes of the partitions and complexity, which reduces the usefulness to homogeneous devices and simple networks associated to IoT. DeeperThings [19], on the other hand, focuses on fully-connected layers and layers where the weights are much bigger than the input activations. This model is restricted to weight input and output partition and also to a specific sizes. One restriction that weight input and output partition has is that the maximum layers possible for layer fusion is fixed to two since after the those two layers the devices have to synchronize. These two frameworks improve from MoDNN since they don't have to explicitly synchronize after each layer, but both of them still lack flexibility since the partition are made for homogeneous devices and simple networks, like YOLOv2 [16], AlexNet [10], etc.

CSDF partitioner [13] performs both model and data partition. To partition, they use weight output partition which restricts the model to synchronize after each layer, which is similar to MoDNN. Since it is focused on throughput, not all the resources are active at the same time and most of the network is restricted to small networks like ResNet18 [6].

Chapter 3

Black-Box Graph Partitioner

3.1 Overview

Fig. 3.1 depicts the overview of BBGraP framework. Firstly, BBGraP will receive the input model and hardware configurations for the naive partitioner (Section 3.3). During the naive partitioning phase, the user will require the BBGraP created previously or the one created through the transformer.

If the user requires a particular partition in any operators, this can be defined in the configuration file, which contains custom partitions of specific layers done by the user. After naive partitioning, the user receive as many network partitions as the number of the devices specified. This divided network can already be transformed in the transformer to be sent directly to the nodes, or those networks can be further processed and optimized in the optimizer module (Section 3.4). The networks be checked in the optimizer module to see if any operator is unnecessary and can be removed. If the user desires, before the transformation back, this network can be sent to the automatic searcher, where BBGraP finds the best partition according to the objective function used in Section 4.4.1. Finally, BBGraP is ready to be transformed

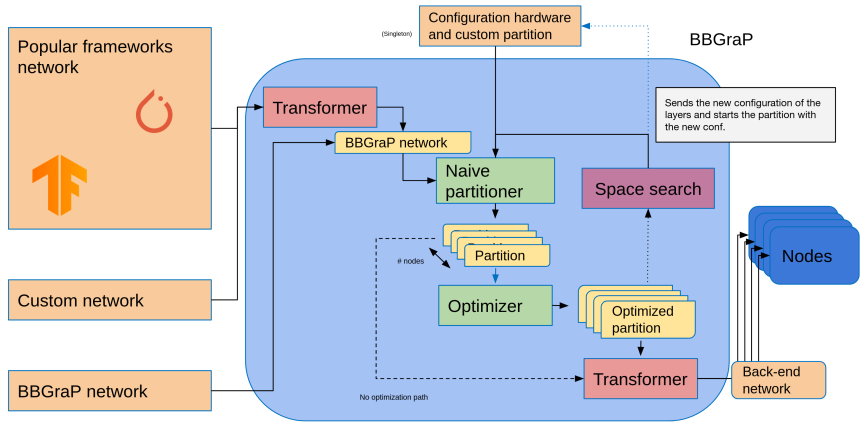


Figure 3.1: Overview of BBGraP framework.

back to the framework the nodes use.

3.2 Transformer

The transformation step is the starting point to get the model from an existing framework. The input model is translated into the network hierarchy shown in Fig. 3.2, where the operators are one of the supported operators in Table 3.1. It should be noted that the arithmetic operators require two inputs, and the dimension of the output should not change from the inputs. Also, operators grouped in 'Others' are the ones that do not require any partitioning.

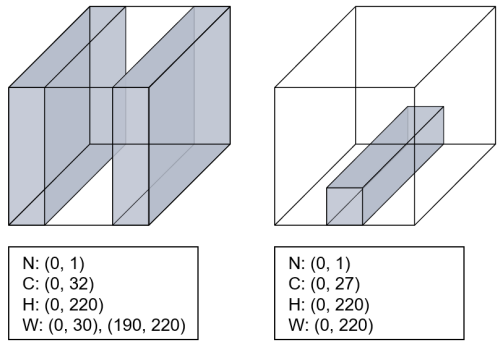


Figure 3.3: ShapeMap examples, where $NCHW$ is number of batches, channels, rows, and columns respectively.

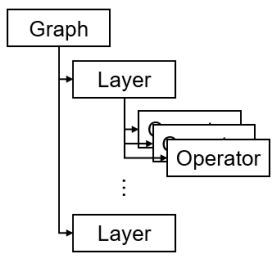


Figure 3.2: Network hierarchy.

Group	**Operators**
Convolutions	Convolution, Group Convolution
Pools	MaxPool, AvgPool, GlobalPool
Arithmetic	Summation, Multiplication
Others	Crop, Concatenation, Activation, Batch Normalization, SoftMax

Table 3.1: Supported BBGraP operators.

The most important part of this step is to get the different feature maps (either weights or inputs) and transform them into our ShapeMap, which later is used for partitioning and optimization. Fig. 3.3 shows a few examples of our ShapeMaps defined in the dimensions of $NCHW$, and each dimension is the directions in which partitioning can occur. Our ShapeMap can even define intervals with multiple ranges, as depicted in the first example of Fig. 3.3.

Finally, the type of partition and the number of devices are also assigned in this transformation step.

Symbol	Description
d	Partitioning direction (c, h, or w)
N_d	Number of devices assigned to direction d
I	Original input size
$pI_{(d,i)}$	Input partition size, partitioned in direction d and assigned to device i
O	Original output size
$pO_{(d,i)}$	Output partition size, partitioned in direction d and assigned to device i
K_d	Kernel size in direction d
tK_d	True kernel size in direction d
D_d	Dilation value in direction d
S_d	Stride value in direction d
IV_d	Interval size in direction d
$P_{(d_0,d_1)}$	Padding size in direction of d_0 (h or w) and d_1 (left or right)

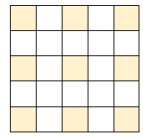
Table 3.2: Table of variables used in the equations

3.3 Naive Partitioner

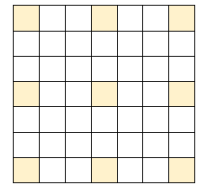
Once the input model is translated to ShapeMaps, we can start with naive partitioning. BBGraP uses the configuration file that contains partition information for a layer, including the number of available devices and the dimensions of each feature map for the partitions. If a configuration file is not given or not used, BBGraP divides the workload evenly across all the devices. Table 3.2 defines the variables used across all the equations in this section.

Naive partitioning is done from bottom to top using the output shape to reference the input. It is easier to know the overlaps due to different stride, padding, and dilation parameters. Following Eq. 3.1, we divide the output size by the number of devices and floor the outcome to partition the outputs evenly to all devices. If any remain after the division, They are distributed in a round-robin manner. For example, 8×8 output feature map partitioned for three nodes in the width-wise division would result in $pO_{(w,0)}$, $pO_{(w,1)}$, and $pO_{(w,2)}$ to be 3×8, 3×8, and 2×8 respectively.

(a) 3×3 kernel with dilation 1.



(b) 3×3 kernel with dilation 2.



(c) 3×3 kernel with dilation 3.

Figure 3.4: 3×3 kernel with different dilation values.

$$pO_{(d,i)} = floor(O/N_d) \tag{3.1}$$

Once BBGraP knows the output dimensions, it creates the input division by taking into account the kernel size, dilation, stride, and padding. First, the proper kernel size is realized with Eq. 3.2. The kernel with dilation 1 is the kernel itself, but with a dilation value higher than 1, the kernel is spread out as shown in Fig. 3.4, thus changing the proper kernel size.

$$tK_d = (K_d - 1) \cdot D_d + 1 \tag{3.2}$$

With the true kernel size found, BBGraP can now calculate the size of the interval from the size of the output partition using Eq. 3.3.

$$IV_{(d,i)} = (O - 1) \cdot S_d + tK_d \tag{3.3}$$

To illustrate the procedure for calculating the interval size, we use Fig. 3.5 that shows a simple example of deriving the interval size 9 when there are 3 output partitions of 1×1, a kernel of 3×3 with dilation and stride of 2.

Given the input interval size, BBGraP can use Eq. 3.4 and Eq. 3.5 to determine

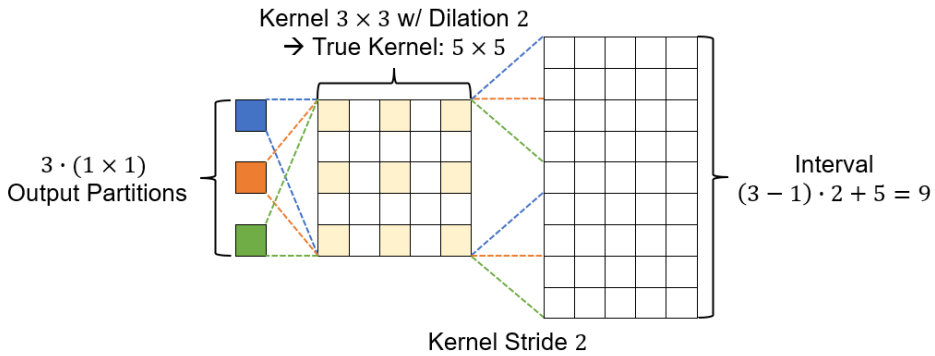


Figure 3.5: Example of calculating input interval size.

the start and end index of the input partition for device i. $P_{(d,0)}$ denotes the padding size in either width-left or height-left, and $P_{(d,1)}$ denotes the padding size in either width-right or height-right. For example, $P_{(w,0)}$ would mean padding size at the left side of the feature map, and $P_{(h,1)}$ would mean padding size at the bottom side of the feature map.

$$\text{Left}_{(d,i)} = \begin{cases} 0 & \text{if } i = 0, \\ S_d \cdot \sum_{j=0}^{i-1} pO_{(d,j)} - P_{(d,0)} & \text{otherwise.} \end{cases} \tag{3.4}$$

$$\text{Right}_{(d,i)} = \begin{cases} IV_{(d,i)} + \text{Left}_{(d,i)} - P_{(d,0)} & \text{if } i = 0, \\ IV_{(d,i)} + \text{Left}_{(d,i)} - P_{(d,1)} & \text{if } i = N_d - 1, \\ IV_{(d,i)} + \text{Left}_{(d,i)} & \text{otherwise.} \end{cases} \tag{3.5}$$

With the values of $\text{Left}_{(d,i)}$ and $\text{Right}_{(d,i)}$, the input partitioned in direction d for device i can thus be defined as Eq. 3.6.

$$pI_{(d,i)} = [\text{Left}_{(d,i)}, \text{Right}_{(d,i)}] \tag{3.6}$$

Once all the partitions are created, BBGraP creates the concatenation and crop

operators to join the outputs of the partitions to keep consistency. This step is the naive partitioning phase; thus, all outputs are concatenated in the direction it was partitioned. This sink creates concatenation operators after every layer. A discrepancy between the concatenated output size and the next layer's input size exist. Therefore, a crop operator is created after all the concatenations for correctness.

3.4 Graph Optimizer

After the naive partitioning phase, BBGraP tries to prune operators or crop data to reduce the number of synchronization points between devices. In order to do that, we compare the dimension of one layer's output to the dimension of the next layer's input, where the comparison results can be distinguished into four different cases, as shown in Fig. 3.6. Fig. 3.6a shows the result of naive partitioning prior to the optimization process, where each device has the process of having the outputs (α), concatenating the output and dependencies from other devices (β), and cropping the concatenated outputs, if necessary, for the next layer (γ). Notice that stage β requires data transfers for concatenation in each device, denoted by the dotted arrow.

Input bigger than the output

If the input is bigger than the output, the necessary data present in other device(s) are fetched and concatenated to the output as shown in Fig. 3.6b. Since BBGraP ensures to bring only the needed data for the next layer instead of bringing the whole output from other device(s) and also eliminate unnecessary crop operation. This optimization, in turn, reduces the data transfer size.

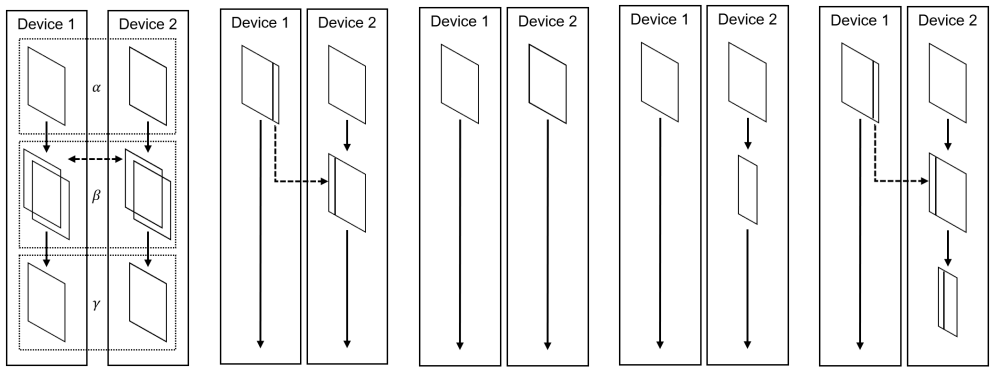


Figure 3.6: Graph optimization methods distinguished into four different cases, where (a) shows the initial state prior to optimization and (b)∼(e) shows the optimization process according to the dimensions of the output and next layer's input.

Input same as the output

When the input is the same as the output, we can delete the crop and concatenation operators since they are unnecessary, as shown in Fig.3.6c.

Input smaller than the output

When the input is smaller than the output, the next layer may or may not need data from other devices. Fig. 3.6d shows the case when we do not need data from other device(s); thus, we can go ahead only with crop operation to pass on to the next layer and eliminate unnecessary data transfers and concatenation operator. However, as shown Fig. 3.6e, there exists a case when we need data from other device(s) and also need to crop for the next layer. In such cases, we combine 3.6b and 3.6d, and BBGraP eliminates unnecessary data transfers from the current device after the concatenation.

3.5 Automatic Search

Once the front end (naive partitioner and optimizer, as seen in previous sections) has been developed, the BBGraP framework is still naive. Because it divides the network layer by layer individually, trying to give the same workload (unless a configuration is given) to the nodes, not taking into account data transfer overhead and only checking that the output is not overlapped. This method sometimes can lead to layer fusion which avoids data transfer, as in the case of the same input and output size explained in Section 3.4. This kind of optimization only using the naive partitioner is more due to the architecture of the network or the configuration given by the user than the framework itself. However, more is needed to find a suitable partition since the data transfer created in the optimization part is still too big to get significant improvement during the inference. Hence this automatic search was added to the framework to find the best case depending on data transfer and heterogeneous speeds of the devices. In this section, it is explained how the implementation has been done. First, the objective function used to find the best partition of the network is described. Right after the objective function, it is explained how the partitions are created and how and why a previous hard pruning has been added. Finally, the last section explains how it has effectively reduced the number of partitions that must be searched; this pruning method, as it is explained, is lossless, so the solution is always the best.

Objective Problem

This section presents the mathematical model of the optimization problem and the reason behind it. This model is rather simplistic and general, containing the most general way of any architecture, and the weights have to be tuned manually with the inference results after running the inference on the existing architecture. Due to

Symbol	Description
l	total number of layers
c	total number of nodes
i	node number
$output\_size$	tuple with the output tensor dimensions of each direction $NCHW$
$crop\_size$	tuple with the optimization crop tensor dimensions of each direction $NCHW$
$dependency\_size$	tuple with the dependencies tensor dimensions of each direction $NCHW$
n_values	Number of values inside the tensor output
$non\_necessary$	Number of overlapped operations
$operations\_w$	weight value of operations value
$data\_w$	weight value of data transfer value
$over\_w$	weight value of the overlapped values
$harmonic\_w$	weight value of the harmonic mean result

Table 3.3: Table of variables used in the equations

the general nature, the model cannot represent with high accuracy the best result. Nevertheless, as seen in the results Section 4, the improvements are up to 2-fold in some networks. The final equation containing the optimization problem's result has been divided into four minor problems: Computation, Data transfer, Overlapped operations, and Fair distribution of the workload. Table 3.3 defines all the variables used during the optimization problem. Starting with the Computation problem in Equation 3.7, using the already known size of each direction, it produces a product of all the directions per each node created. For example, in the case, the operator has an output $output\_size$ of $1 \times 3 \times 5 \times 5$ the result for n_value would be 75.

$$n\_value_i = \prod(output\_size_i) \tag{3.7}$$

From the list of product, sizes which is the total size of the tensor, n_values created in Equation 3.7, a maximum value has to be selected named $n\_operations$. This value is the bottleneck to execute the next layer in case no data transfer is involved, which would need further synchronization. The maximum value selection is made

in Equation 3.8. This value is multiplied by the weight called $operation\_w$ that the user previously defined. Finally, this value is used in the final equation to find the total optimization problem value.

$$n\_operations = max([n\_value_0, n\_value_1, ..., n\_value_{c-1}]) * operations\_w \quad (3.8)$$

A simple way to find the data transfer is to use the data dependencies created during the optimization (Section 3.4), which is always performed before the objective function is called. Equation 3.9 calculates the data transfer in the different nodes and use the $dependency\_size$, which is the output size of the dependency operator, to find the total number of value transfers in that layer. Later this value is added to the total optimization problem value. $data\_w$ is also a user-defined weight like $operation\_w$.

$$data\_trans = \sum_{i=0}^{c-1} \sum_{j=0}^{d} \prod dependency\_size_{i,j} * data\_w \quad (3.9)$$

Once the values of the data transfer and the total number of operations have been found, the following key value to find is if the partition has overlapped operations, Equation 3.10. For this, it is used the crop operations created during the optimization (Figure 3.6d-3.6e). This crop operation indicates that some operations are already calculated in another node, leading to a non-optimal partition.

$$non\_necessary = \sum_{i=0}^{c-1}(\prod crop\_input\_size_i - \prod crop\_output\_size_i) * over\_w \quad (3.10)$$

The harmonic mean is the last value to calculate before adding them all together. This value calculated in Equation 3.11 is added to penalize the partitions with an unbalanced workload.

$$harmonic = \left| n\_operations - \frac{c}{\sum_{i=1}^{c} \frac{1}{n\_value_i}} \right| * harmonic\_w \quad (3.11)$$

Finally, all the values calculated in the Equations 3.8 to 3.11 are added together from all the layers in Equation 3.12. This value is used later during the pruning face to distinguish between valid and not valid partitions in Section 3.5 that have a more detailed explanation of the process.

$$value = \sum_{i=0}^{l}(harmonic + data\_trans + n\_operations + non\_necessary) \quad (3.12)$$

Partitions creation

For the creation of the partition, since we cannot create all the partitions and store them in the memory, we must develop a method fast enough to create an indefinite

Symbol	Description
d	Partitioning direction (c, h, or w)
n	total number of nodes
$size\_output$	size of the output tensor
$threshold$	configuration value to prune the partitions

Table 3.4: Table of variables used in the equations

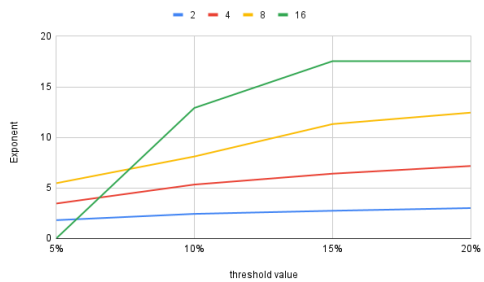


Figure 3.7: Log based result of the total number of possibilities per number of nodes and threshold pruning in the width direction of 1x3x224x224.

number of partitions for scalability. As seen in Figure 3.7 the number of possibilities are too big and they increase exponentially for every node it is added. The solution is to create the partitions procedural. Instead of storing all the partitions, store only the intervals and a linking dictionary of those intervals to link them later together and make the partitions. This method allows the framework to store all the partitions, no matter how many they are, and the execution is almost instantaneous. Table 3.4 shows the variable that is used for the equations of this section.

The first step is to get the interval sizes for each node depending on the speed of those nodes. Following the Equation 3.13, the $interval\_size$ possible, which reflects the size of the partition per node, are created according to the different speeds of the nodes before using them to create the actual intervals of each node. Looking at Figure 3.10, this would be 1 for nodes 1 and 3 and 4 for node 2. The difference in the interval values is due to the speed where node 2 is 3$\times$ faster than nodes 1 and 2.

$$interval\_size_j = \lfloor size\_output_d / \sum_{i=1}^{n} speeds_i * speeds_j \rfloor \tag{3.13}$$

Once the $interval\_size_j$ is created, the minimum and maximum sizes should be determined. These limits are affected by a hard value determined by the user named $threshold$. This value is used to no search across all existing partitions since the best solution always falls not far away from the $interval\_size_j$. The minimum size named $start$ and maximum named end are created in the Equations 3.14 and 3.15. These values would be $[0, 2]$ for nodes 1 and 2 and $[4, 5]$ for node 2 if the example to find the $interval\_size$ from the previous paragraph is used, as shown in Figure 3.10.

$$start_j = \lceil (interval\_size_j(100 - threshold)/100 \rceil \tag{3.14}$$

$$end_j = \lceil (interval\_size_j(100 + threshold)/100 \rceil \tag{3.15}$$

Finally, the list with all the possible interval sizes can be created after knowing the minimum and maximum values. Starting at $start_j$ and finishing at end_j increasing the value by one. The list's creation can be seen in Equation 3.16.

$$possible\_sizes_j = [start_j, \ldots, end_j] \tag{3.16}$$

Symbol	Description
$ds_{core,start}$	intervals sorted by node and start value
$de_{core,end}$	intervals sorted by node and end value
$link$	relates the partition of one node to another
$nodes$	number of nodes in total
$total$	total number of partitions

Table 3.5: Table of variables used in the equations

Now that the $possible\_sizes$ per node are known creating the dictionaries to build later the partitions is explained. First, this object contains the fields listed in Table 3.5.

To create the partitions for each node, this should be separated into 3 different groups:

Left node

This first node is the node that contains the initial intervals of the tensor in the chosen direction. This node is also associated to the node value of 0 for simplification.

$$ds_{0,0} = [(0, possible\_size_{0,0}), (0, possible\_size_{0,1}), \ldots, (0, possible\_size_{0,N})] \tag{3.17}$$

$$de_{0,possible\_size_{0,i}} = (0, possible\_size_{0,i}) \tag{3.18}$$

In this node, the ds only contains one start index and its list since all the intervals start at 0. The de have to create an end index for each interval and store only one

interval per each index since there are not repeated endings index for the starting node because all of them start at 0. If there were a repeated end index, it would mean a repeated interval, which would be an error. Using the example in Figure 3.10 and the results previous obtained ($[0, 1, 2]$) the intervals are $(0, 1)$ and $(0, 2)$.

Right node

Like the previous left node, this node repeats a similar pattern on the other side, inverting the ds and de way of creation. This inversion is because this node contains the last interval, and the fixed index, in this case, is the ending index corresponding to the size of the output or $output\_size$. For simplification, this node has the value of n corresponding to the last node or $num\_nodes - 1$ on the Equations 3.19, 3.20 and 3.21.

$$diff_i = output\_size - possible\_size_{n,i} \tag{3.19}$$

$$ds_{n,diff_i} = (diff_i, output\_size) \tag{3.20}$$

$$de_{n,output\_size} = [(diff_0, output\_size), \ldots, (diff_N, output\_size)] \tag{3.21}$$

The most remarkable difference compared to the left node creation is the Equa-

tion 3.19. Since the output size and the list of possible interval sizes are only known, the left index has to be found through the difference between those variables. This equation calculates the left index for each rightmost node partition; in our example, this would be 6 and 7, creating the intervals $(6, 8)$ and $(7, 8)$.

Middle nodes

Since there is no clear reference on where to start or end in the middle nodes intervals, it should be created with the other interval sizes from other nodes. The creation of those references is in the Equations 3.22 and 3.23, where it adds all the smallest sizes and all the biggest sizes and subtracts them to the $size\_output$ to create the upper (max_idx) and lower (min_idx) limit for the starting index.

$$min\_idx_n = size\_output - \sum_{i=node}^{num\_nodes} possible\_size_{i,N} \tag{3.22}$$

$$max\_idx_n = size\_output - \sum_{i=node}^{num\_nodes} possible\_size_{i,0} \tag{3.23}$$

To understand clearly how these intervals are created, Figure 3.8 has an example of the previous examples. In this Figure the min_idx_n is $8 - (5 + 2) = 1$ and max_idx_n is $8 - (4 + 1) = 3$ where the node is 1 so only 2 nodes are added together.

Once the upper and lower limit is created, the possible starting and ending indexes can be stored in $'ds'$ and $'de'$. Using the already known results from the previous example, the list of indexes $[1, 2, 3]$ corresponding to $shift\_idx_n$ can be obtained, which are the possible starting index of the intervals.

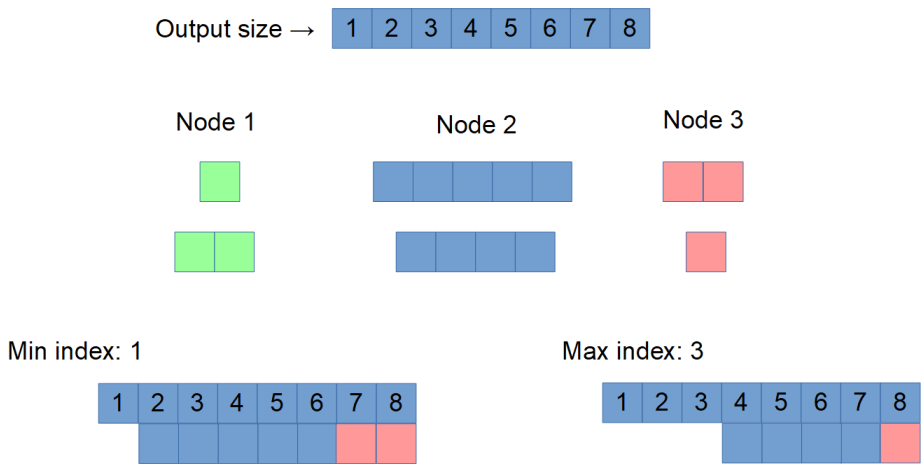


Figure 3.8: Example of how the min_idx_n and max_idx_n are found.

$$shift\_idx_n = [min\_idx_n, ..., max\_idx_n] \tag{3.24}$$

The interval values in $shift\_idx_n$ are used as $start\_idx$ for the middle nodes and find $ending\_idx$; the interval sizes have to be added to each $start\_idx$. This operation created a matrix of endings like in Equation 3.25. Furthermore, a graphic example of the intervals constructed can be seen in Figure 3.9. One thing to remark is that only some endings are possible neither are all the starts. The not allowed starts and ends are the ones that correspond to 0 and the $output\_size$, like the interval $(3, 8)$ shown in the graphic example. Other intervals are discarded in the linking part, as it is explained in Section 3.5.

$$shift\_end_{n,i,j} = \begin{bmatrix} shift\_idx_{n,0} + possible\_size_{n,j} & \dots & shift\_idx_{n,i} + possible\_size_{n,j} \\ \vdots & \ddots & \vdots \\ shift\_idx_{n,0} + possible\_size_{n,0} & \dots & shift\_idx_{n,i} + possible\_size_{n,0} \end{bmatrix} \tag{3.25}$$

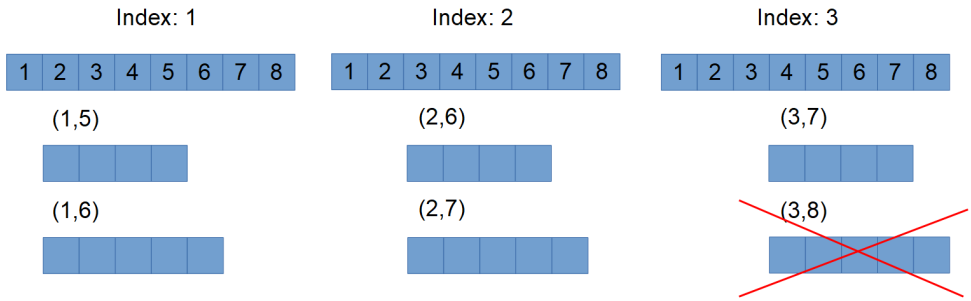


Figure 3.9: Example of how the middle nodes are created.

Linking partitions

Linking partitions is a dictionary that, instead of storing a $start\_idx$ related to an interval, it relates the $start\_idx$ of one interval of one node to each interval of the next node. How this is accomplished is by checking if the next node $start\_idx$ is higher than the $start\_idx$ of the current node interval, it is also checked it also checks if the $start\_idx$ of the next node interval is equal (or lower in case the overlapped option is activated) to the end_idx. Overlapped case means when the partitions are allowed to have intervals that are already in another node. An example of an overlapped case is in Figure 3.10, where the red color means overlapped values. Another example using the intervals of Section 3.5 and 3.5 is using the interval $(0, 2)$ from node 1, and with the previous statements, a valid interval would be $(1, 5)$, but a non-valid interval for this previously selected interval would be $(3, 6)$.

In Figure 3.10, there is an example showing the pruning improvement and all the possible partitions after the creation of the dictionaries. In the example the $size\_output$ is 8 and the $speeds$ are $[1, 3, 1]$ and finally the $threshold$ is 20%. The total number of partitions is reduced from 176 to 5, equivalent to $35.2x$.

Table 3.6, 3.7, 3.8 and shows an example with a different $output\_size$ and ho-

Dictionary	Node	Index	Partitions
ds	0	0	(0,2) (0,3) (0,4)
	1	1	(1,3) (1,4) (1,5)
		2	(2,4) (2,5) (2,6)
		3	(3,5) (3,6) (3,7)
		4	(4,6) (4,7) (4,8)
		5	(5,7) (5,8)
	2	7	(7,9)
		6	(6,9)
		5	(5,9)

Table 3.6: Example of the dictionary $'ds'$

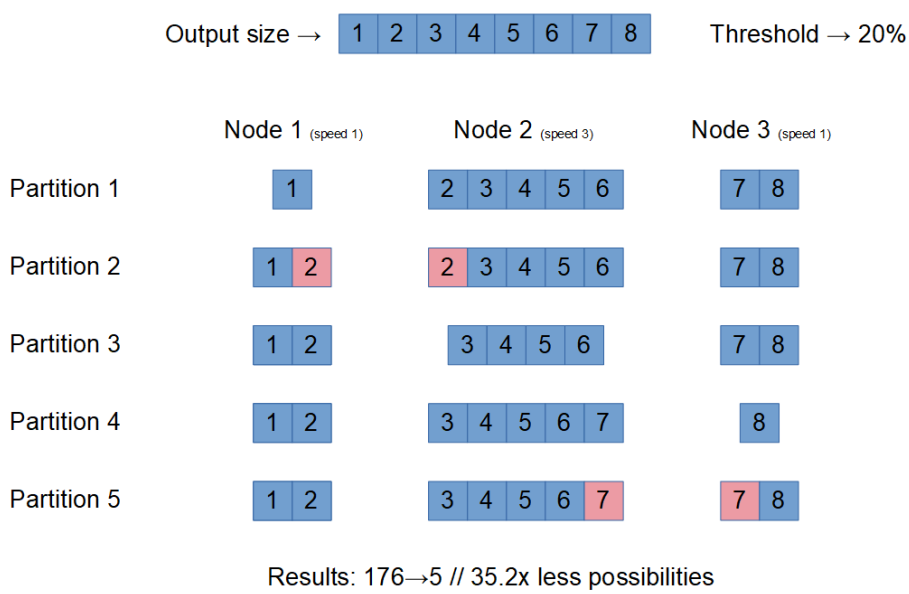


Figure 3.10: Example of the reduction from the $threshold$.

Dictionary	Node	Index	Partitions
de	0	2	(0,2)
		3	(0,3)
		4	(0,4)
	1	3	(1,3)
		4	(1,4) (2,4)
		5	(1,5) (2,5) (3,5)
		6	(2,6) (3,6) (4,6)
		7	(3,7) (4,7) (5,7)
		8	(4,8) (5,8)
	2	9	(5,9) (6,9) (7,9)

Table 3.7: Example of the dictionary $'de'$

Dictionary	Node	End index	Next Start
$link$	0	2	1,2
		3	1,2,3
		4	1,2,3,4
	1	3	
		4	
		5	5
		6	5,6
		7	5,6,7
		8	5,6,7

Table 3.8: Example of the dictionary $'link'$

mogeneous speeds of the dictionary once it is finished. In Table 3.6 it can be seen the points remarked from the Equations 3.17 and 3.20. Where on the node number 0, it can be seen that there is only one index, while on the node number 2, which in this case is the right node or last, it can be seen that there is an index for each partition. This pattern are also found but inverted in the table 3.7. In both of these tables, it can be seen that in the middle nodes, only some indexes have the same number of partitions. Also, if the $shift\_end_{n,i,j}$ is creating a none possible partition $(1, 4)$ or $(1, 3)$ that as can be seen in the linking Table 3.8 is not used to link since the index 4 is not found in the node number 1.

	Naive	BBGraP
$Time$	$O(M^N)$	$O(M \times N)$
$Memory$	$O(M^N)$	min $O(N)$, max $O(M \times N)$

Table 3.9: O notation comparison of the creation and storage

	Naive	BBGraP
$Time$	$O(1)$	min $O(N)$, max $O(M \times N)$

Table 3.10: O notation comparison of how fast is to search a partition in different methods

This method can also access random partitions only using an ID in $O(N)$, where N is the number of nodes. Nevertheless, this method is out of the scope of this thesis since the method, even though it has been tested, is not used during the pruning method. Table 3.9 and 3.10 show the difference between the method previously explained and the most naive of creating and storing all the partitions where N is the number of nodes and M is the size of the output in the direction of the partition.

As can be seen, the memory footprint and time to create the partitions have been significantly reduced from exponential to linear. In exchange for those improvements, the search of the partitions is no longer $O(1)$, but it became linear. Even with this change, the improvement in the creation and memory makes the delay in search worthy since it allows the algorithm to create partitions even for large tensors and many nodes in a short time.

Partitions pruning

Now that the process of how the partitions are created is understood. It is described how the best partition is selected to minimize the cost of the optimization problem. Firstly, it is explained how the partitions are ordered. Secondly, it is described how the one better between the partitions is selected. Finally, the method used to prune

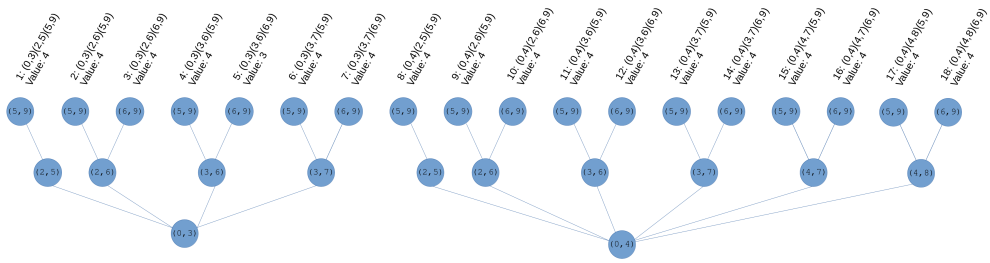


Figure 3.11: Graphic representation of the partitions creation.

and not try all the partitions since could take an incredible amount of time. During this process, it only considers one partition layer for simplification.

The Algorithm 1 is used to understand how the algorithm orders the partitions. In this algorithm, it can be seen first that it creates the $current\_p$ in Line $2-7$, which is used to store the intervals of each node inside. These lines also get the links between the previous and current nodes. This recursive procedure is done to go across the intervals inside the $'ds'$ dictionary in the second loop in Line 17. Once the list is created, the algorithm loops through the links list and store the intervals of the current node based on the $start\_idx$ used in Line 10. Finally, if the node is the last node possible, it calculates the optimization problem after dividing the layer according to the $tmp\_partition$ selected and store it into $results$ in Line $16-18$. If the node is not the last, it recursively goes to the next node in Line 14.

In Figure 3.11 also can be seen that the tree is created using the dictionary of Table 3.6, 3.7, and 3.8. Some of the intervals have been removed to keep it more simple and compact. The removed intervals doesn't effect the results neither the pruning algorithm. By the recursive algorithm, it can also be seen how the algorithm groups the trees by $start\_idx$ and in order. This example is important later in the pruning section.

After procedurally creating the partitions using the previously created dictionary in Section 3.5, the algorithm should be able to find the best result in all those parti-

Algorithm 1 Recursive partitions evaluation

```
1: procedure PARTITIONSEARCH(node, current_p)
2:     if current_p is None then
3:         list_possible_start ← [0, ]
4:         tmp_partition ← [0, ] × num_nodes
5:     else
6:         list_possible_start ← partitions['link']
   [node − 1][current_p[−1][−1]]
7:         tmp_partition ← current_p
8:     end if
9:     for start_idx in list_possible_start do
10:        list_intervals ← partitions['ds'][node][start_idx]
11:        for interval in list_intervals do
12:            tmp_partition[core] ← interval
13:            if node ≠ num_nodes − 1 then
14:                results ← PARTITIONSEARCH(node + 1, tmp_partition)
15:            else
16:                layer ← LAYERCREATION(tmp_partition)
17:                opt_value ← OPTIMIZATION_PROBLEM(layer)
18:                return results[tmp_partition] ← opt_value
19:            end if
20:        end for
21:    end for
22:    return results
23: end procedure
```

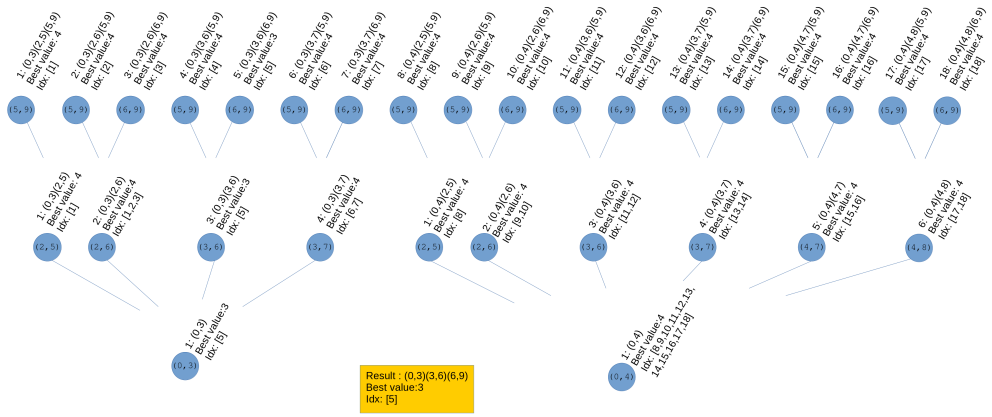


Figure 3.12: Graphic representation of the partitions creation with the best result.

tions. Using the previous Algorithm 1 with a couple of additional lines included in the Algorithm 2, the best result can be found.

As it can be seen from Algorithm 2, there are two types of best values one for the trees with the same start and one for all the trees in the same node. The best value for the tree is stored inside in $best\_res\_tree$ in the Lines $24 - 29$. Moreover, the best value for the node is stored in $best$ Lines $31 - 36$, which is the value returned to previous nodes. Doing this kind of storage, only the best values are sent to the previous node making 100% the result always the best.

In Figure 3.12, differently from Figure 3.11, it can be seen how it keeps track of the best value across all the nodes.

Finally, Algorithms 3, 4, and 5 show how the partitions are pruned with the help of the previously explained algorithms. Starting with Algorithm 3 show how the bad partitions that is not calculated again are chosen. These partitions are chosen based on the best partition in that tree. In Lines $4 - 6$, intervals from the next node have been calculated together with the interval of the current node. In the following Lines $7 - 10$, the found intervals are removed from the $bad\_partitions$. The remaining

Algorithm 2 Recursive partitions evaluation to find best

procedure PARTITIONSEARCH($node, current\_p$)
  if $current\_p$ is None **then**
    $list\_possible\_start \leftarrow [0,]$
    $tmp\_partition \leftarrow [0,] \times num\_nodes$
  else
    $list\_possible\_start \leftarrow partitions['link']$
  $[node-1][current\_p[-1][-1]]$
    $tmp\_partition \leftarrow current\_p$
  end if
  $best \leftarrow \infty$
  $best\_list \leftarrow None$
  for $start\_idx$ in $list\_possible\_start$ **do**
    $list\_intervals \leftarrow partitions['ds'][node][start\_idx]$
    $best\_res\_tree \leftarrow \infty$
    $best\_tree\_list \leftarrow None$
    for $interval$ in $list\_intervals$ **do**
      $tmp\_partition[node] \leftarrow interval$
      if $node \neq num\_nodes - 1$ **then**
        $best_list_tmp, best_tmp \leftarrow$PARTITIONSEARCH($node + 1$,
  $tmp\_partition$)
      else
        $layer \leftarrow$ LAYERCREATION($tmp\_partition$)
        $best\_tmp \leftarrow$ OPTIMIZATION_PROBLEM($layer$)
        return $best\_list\_tmp[tmp\_partition] \leftarrow opt\_value$
      end if
      if $best\_tmp < best\_res\_tree$ **then**
        $best\_res\_tree \leftarrow best\_tmp$
        $best\_tree\_list = best\_list\_tmp$
      else if $best\_tmp = best\_res\_tree$ **then**
        $best\_tree\_list.extend(best\_list\_tmp)$
      end if
    end for
    if $best\_res\_tree < best$ **then**
      $best \leftarrow best\_res\_tree$
      $best\_list = best\_tree\_list$
    else if $best\_res\_tree = best$ **then**
      $best\_list.extend(best\_tree\_list)$
    end if
  end for
  return $best\_list, best$
end procedure

intervals and the tree coming from them is not calculated if it is found again. The question that arises doing this is how it is possible to know in advance that those intervals do not contain a better result than the optimal one. First, it must be noticed that the trees being compared are the ones starting at the same index. Making this assumption, these trees can be isolated and say that the results independent of the intervals in previous nodes always contain the same results. For example, if there is a partition $(0,3)(2,5)(5,9)$ and a partition $(0,4)(2,5)(5,9)$ and finally $(0,2)(2,5)(5,9)$. With these partitions and using the interval $(2,5)$, which would be in the tree starting at index 2, it can be seen that the partitions $(2,5)(5,9)$ consistently reproduce the same results. These results are independent of the previous intervals $(0,2)$, $(0,3)$, and $(0,4)$. So if one of these cases is worse than the previous one, it can be safely assumed it has not a better result in the future, and thus it can be deleted. This example can be seen in Figure 3.14.

In Algorithm 4, it is compared to the current interval and checks if that interval exists in the $dict\_skip\_intervals$. If the node is previous to the current one, the algorithm only has to check if that interval is in the $dict\_skip\_intervals$. However, suppose the node is the same as the current one. In that case, it has to check if the interval from the previous node is in the $dict\_skip\_intervals$ and if it exists, check if the interval of the current node is inside the list corresponding to the interval of the previous node. If any of these conditions are right, the interval is skipped.

Finally, Algorithm 5 adds the Algorithm 4 and 3 into the previous Algorithm 2 in Line 16 and 45. It also added a break of the loops in case the best case found is worse than the previous one in Lines 33 and 42.

This pruning can be accomplished thanks to the recursive ordering of the partitions, and the objective function explained in Section 3.5. As it can be seen in Figure 3.13, the optimization problem, since it brings the best result to the top as seen in

Algorithm 3 Store partitions that won't be searched again

```
1: procedure StoreBadPartitions(bad_partitions, partitions,
   interval, best_tree_list, start_idx, node)
2:     start_next ← partitions['link'][node][interval[−1]]
3:     tmp_all_intervals ← []
4:     for start_tmp in start_next do
5:         tmp_all_intervals.extend(partitions['ds'][node + 1][start_tmp])
6:     end for
7:     for best_part in best_tree_list do
8:         if best_part[node + 1] in tmp_all_intervals then
9:             tmp_all_intervals.remove(best_part[node + 1])
10:        end if
11:    end for
12:    bad_partitions[node][start_idx]['base'] = tmp_all_intervals
13:    for best_part_idx in best_list_tmp do
14:        for node_tmp in [node + 1, . . . , num_nodes − 1] do
15:            start_next ← partitions['link'][node_tmp]
   [best_part_idx[node_tmp][−1]]
16:            tmp_all_intervals ← []
17:            for start_tmp in start_next do
18:                tmp_all_intervals.extend(partitions['ds']
   [node_tmp + 1][start_tmp])
19:            end for
20:            for best_part in best_tree_list do
21:                if best_part[node + 1] in tmp_all_intervals then
22:                    tmp_all_intervals.remove(best_part[node + 1])
23:                end if
24:            end for
25:            bad_partitions[node][start_idx][node_tmp]
   [best_part_idx[node_tmp]].extend(tmp_all_intervals)
26:        end for
27:    end for
28: end procedure
```

Algorithm 4 Store partitions that won't be searched again

```
1: procedure SKIPPARTITION(dict_skip_intervals, node, interval, current_p)
2:     skip_interval ← false
3:     for node_tmp, list_intervals in dict_skip_intervals do
4:         if node − 1 = node_tmp then
5:             if interval in list_intervals then
6:                 skip_interval ← true
7:             end if
8:         else if node ≠ node_tmp and
   currrent_p[node − 1] in list_intervals and
   interval in list_intervals[current_p[node − 1]] then
9:             skip_interval ← true
10:        end if
11:    end for
12:    return skip_interval
13: end procedure
```

Figure 3.12, it created a convex function without local minimums only absolute maximums. It can be seen that this is true for any group of partitions, size of output, or node distribution because the partitions are constantly increasing in size.

Since some sizes are too small at the beginning, the other requires to be bigger than it should be to create a valid partition. Furthermore, while increasing the sizes, the algorithm finds balance. After finding the one that is in balance, the value of the objective function also gets worse due to one of the sizes being too big; usually, that interval is the one in node 0. In the example in Figure 3.14, it can be seen that after the partition $(0, 3)(3, 6)(6, 9)$, none of the subsequent partitions are better than that one neither it creates a local minimum.

Algorithm 5 Recursive partitions evaluation to find best

for $interval$ in $list\_intervals$ **do**
    if SKIPPARTITION($dict\_skip\_intervals$, $node, interval, current\_p$) $=$ **true then**
        $continue$
    end if
    $tmp\_partition[node] \leftarrow interval$
    if $node \neq num\_nodes - 1$ **then**
        $best\_list\_tmp, best\_tmp \leftarrow$ PARTITIONSEARCH($node + 1$, $tmp\_partition$)
    else
        $layer \leftarrow$ LAYERCREATION($tmp\_partition$)
        $best\_tmp \leftarrow$ OPTIMIZATION_PROBLEM($layer$)
        return $best\_list\_tmp[tmp\_partition] \leftarrow opt\_value$
    end if
    if $best\_tmp < best\_res\_tree$ **then**
        $best\_res\_tree \leftarrow best\_tmp$
        $best\_tree\_list = best\_list\_tmp$
    else if $best\_tmp = best\_res\_tree$ **then**
        $best\_tree\_list.extend(best\_list\_tmp)$
    else
        break
    end if
end for
if $best\_res\_tree < best$ **then**
    $best \leftarrow best\_res\_tree$
    $best\_list = best\_tree\_list$
else if $best\_res\_tree = best$ **then**
    $best\_list.extend(best\_tree\_list)$
else
    break
end if
if $node \neq num\_nodes - 1$ **then**
    STOREBADPARTITIONS($bad\_partitions, partitions, interval$ $, best\_tree\_list, start\_idx, node$)
end if

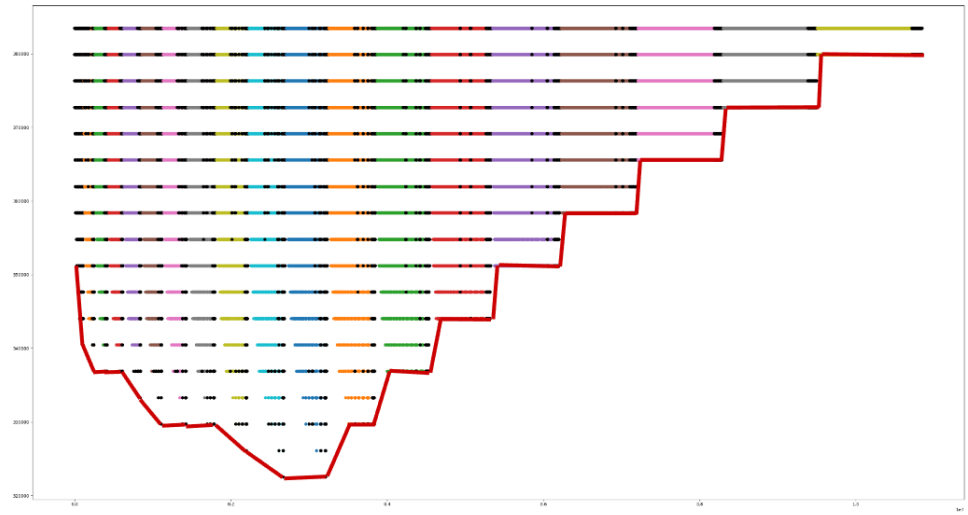


Figure 3.13: Scatter plot of each partition where the colors mean the tree of each interval in node 0 and the red line are the best results of trees and the black points are the actual calculated partitions after pruning.

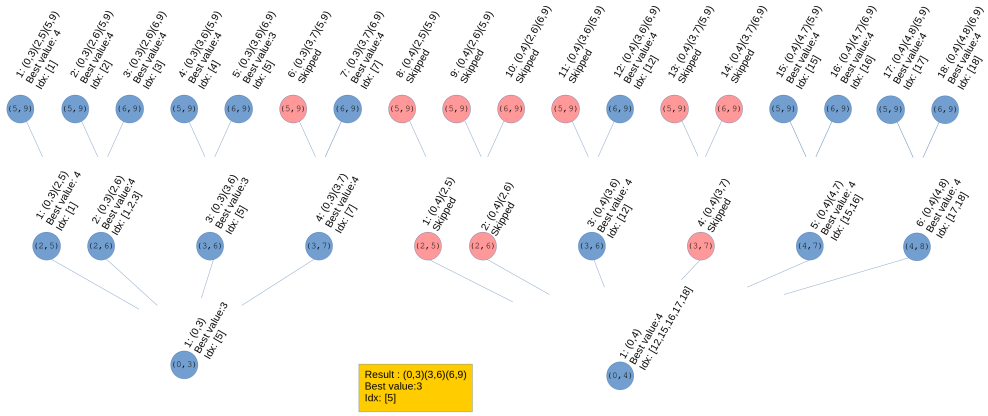


Figure 3.14: Graphic representation of the partitions creation with pruned partitions.

Chapter 4

Evaluation

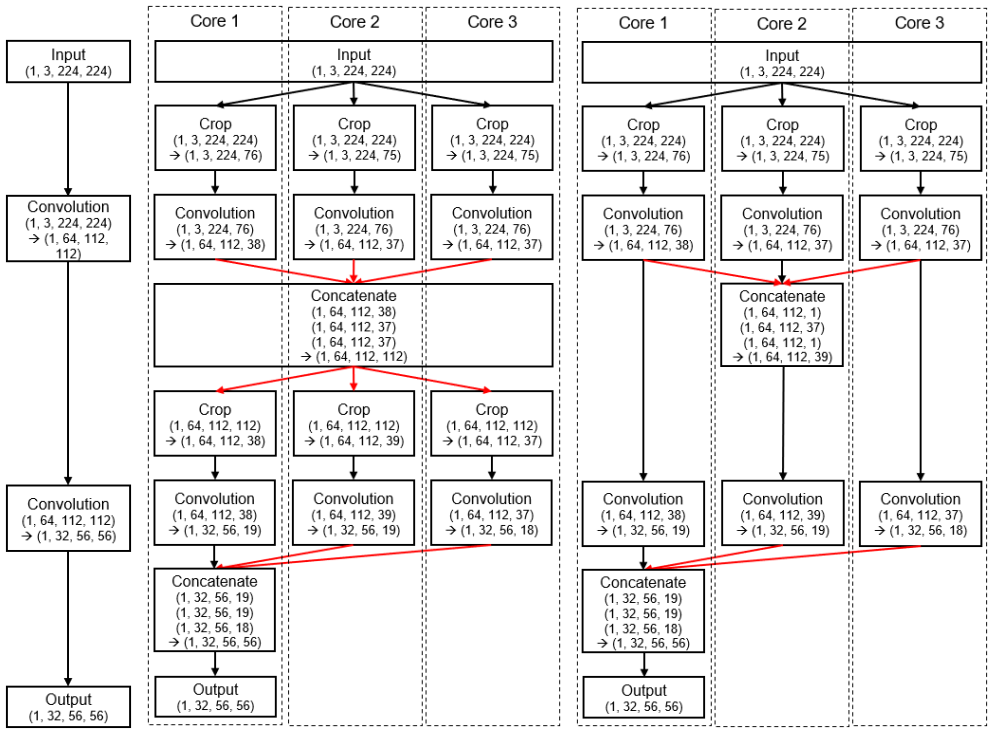
4.1 Graph Partitioning and Optimization

This section shows an example of what happens to the input model as it goes through the graph partitioning and optimizing phase with a small example network.

Fig. 4.1 shows the whole process of BBGraP with a small network consisting of two convolution layers, partitioning into three nodes. Fig. 4.1a shows the original graph, and Fig. 4.1b shows that the first phase of naive partitioning has divided the graph equally. However, the first phase created several operators with few being redundant; thus, the graph optimization stage eliminates said redundant operators, where in some cases, it even achieves layer fusion. The final output of the graph optimization stage is depicted in Fig. 4.1c. All data dependencies in the final graphs are marked with red arrows.

4.2 Pruning results

To evaluate how the pruning method improves, a convolution operator with an output tensor of $1 \times 3 \times 448 \times 488$ is used. To see if the pruning method is excellent and

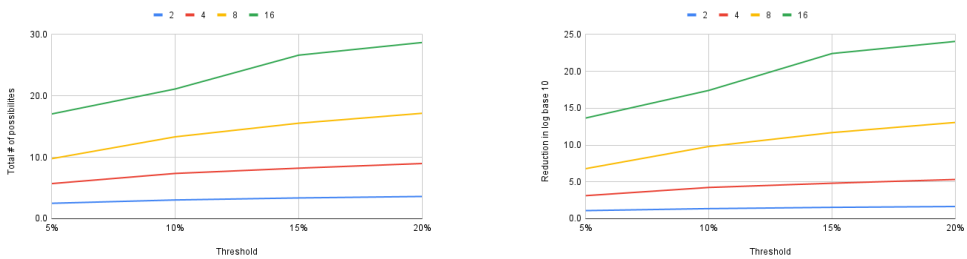


(a) Original (b) After naive partitioning stage (c) After graph optimization stage

Figure 4.1: Naive partitioning and graph optimization example. Red arrows involve data transfers between the cores for concatenation operator.

# nodes	Threshold (%)	5	10	15	20
2	Total (log10)	2.5	3.1	3.4	3.6
	Pruned	26	48	70	92
	Reduction (log10)	1.1	1.4	1.5	1.7
4	Total (log10)	5.7	7.4	8.2	9
	Pruned	389	1355	2600	4622
	Reduction (log10)	3.1	4.2	4.8	5.3
8	Total (log10)	9.8	13.3	15.5	17.2
	Pruned	966	3417	7344	12747
	Reduction (log10)	6.8	9.8	11.7	13.1
16	Total (log10)	17.1	21.1	26.6	28.7
	Pruned	2501	5174	16528	43121
	Reduction (log10)	13.7	17.4	22.4	24.1

Table 4.1: Results for different $thresholds$ in the top row and different nodes in the left column



(a) Original number of possibilities.

(b) Reduction from the total possibilities.

Figure 4.2: Results of the pruning method

scalable, it is used up to 16 nodes, and the $threshold$ value is also changed to ensure it is working, changing all the parameters. The results of how these parameters affect the improvement of the pruning can be seen in Table 4.1 and Figure 4.2.

Table 4.1 shows how it would be impossible to evaluate all the partitions since when the nodes are increased to only 4 nodes and a threshold of 20% the number of possibilities is already billions of them. This number only increases exponentially with more nodes added, and the more the $threshold$ value is increased. However, with the method of pruning the partitions, the number of partitions calculated are

under $50,000$, and the most important thing is that it increases linearly with both variable; nodes and $threshold$. The reduction also increases exponentially due to the nature of the partitions tree. Since the partitions tree contains more repeated trees than when the number of nodes and $threshold$ value is small produces a higher pruning.

4.3 MIDAP results

In this section, we evaluate how BBGraP works with different configurations and different given graphs. To do this evaluation, MIDAP [8] [9], a multi-core fully pipelined execution FPGA, has been used. In order to do the evaluation, we created a transformer to translate from MIDAP to BBGRAP and translate back the divided graph. Since the operators for the synchronization are implemented in the simulator, no runtime or compiler modification has been done. One restriction with this simulator is that the tensor can only divide on its height direction. As shown in the results, this has some overhead and reduces the inference's improvements. Another limitation we have is that the simulator for multi-core is still under development, so only a limited of networks are available for multi-core.

First, an analysis of two popular networks will be done to know the nature of the most common CNN used on MIDAP. After the analysis, the results of all the working networks on MIDAP multi-core will be explained. Furthermore, an analysis of how the optimization improves compared to the naive approach will be shown. Unfortunately, because MIDAP is still under development, the improvement of using the automatic search will be shown in the next section due to the inability of the simulator to run the created network.

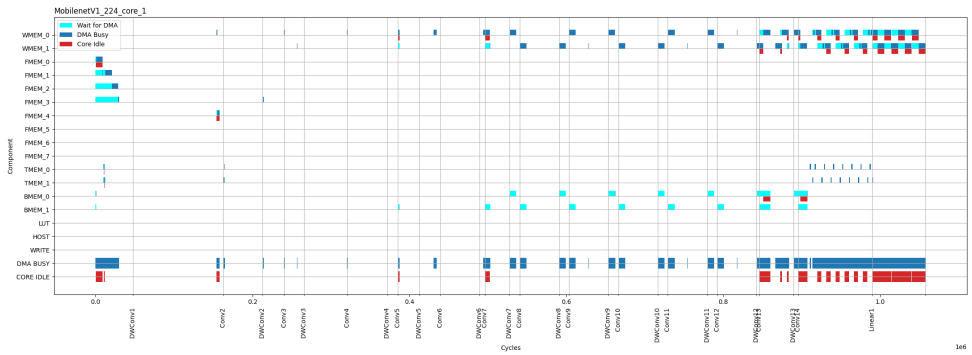


Figure 4.3: $Mobilenet$ inference profiler baseline.

4.3.1 Mobilenet Network Analysis

Since the results have two significant groups, this $Mobilenet$ [7] and $ReseNet$ [6], this has been selected to see the significant characteristics that will affect the results. Figure 4.3 shows the profile of the $Mobilenet$ inference on MIDAP without division.

To understand the profiler, some labels of the figure should be explained before. WMEM is referred to the memory that stores the weight values, FMEM is the feature map memories, TMEM is the write off-chip buffers, BMEM is the memories to store the bias values, DMA busy shows when the memory bus that connects the on-chip and off-chip memory is busy, the core idle shows in red when the cores are not doing operations and finally LUT, Host and Write are not used. Another thing to remember is the difference between light blue and dark blue. Light blue is the memory block waiting for another memory to finish the load. Moreover, dark blue is when the memory is busy loading from the off-chip.

In the $Mobilenet$ profile in Figure 4.3, it can be seen that almost the whole network fits into the memory since the first feature map is the biggest one in the network. This type of network, where the biggest feature map fits into memory, will have a reduced improvement during the division. Since the overhead of the off-loading and on-loading feature maps to transfer to other cores will be more noticeable compared

to zero transfers of the baseline.

Other noticeable characteristics are the different types of layers that can be classified into three more general groups that can also be seen in Figure 4.4:

- Feature map bigger than weight map in the case of $Mobilenet$ are layers $Conv1$ to $Conv7$.
- Feature map similar to weight map in the case of $Mobilenet$ this is layers $Conv7$ to $Conv13$.
- Feature map smaller than weight map in the case of $Mobilenet$ are layers $Conv13$ to $Linear1$.

From the performance in Figure 4.4, it can be seen how these groups of layers affect the performance during the division on the naive case. In Group 1, the layers perform as they should, with a speedup of around $2\times$ in all the layers. Except for the $DWConv1$ that, due to a simulator error, slows it down $0.5\times$. For Group 2, the performance is reduced due to the increased weight size; in some cases, the data transfer between cores in the layers slows down. In the last Group 3, the performance is worse than that of three cores due to the increase in the weight size. In this last group, there are also two exceptions $GlobalPool1$ that in the baseline is not performed since it is pipelined. And the last exception, $Linear1$, that being FC, is not possible to divide by height direction.

Some of this overhead due to the groups could be solved if MIDAP allowed channel direction division since it would have smaller weight maps. Nevertheless, that method also has shortcomings in that it should synchronize after each layer unless further optimization exists. However, this shortcoming could be overlooked by the improvement of performance obtained by the channel direction division.

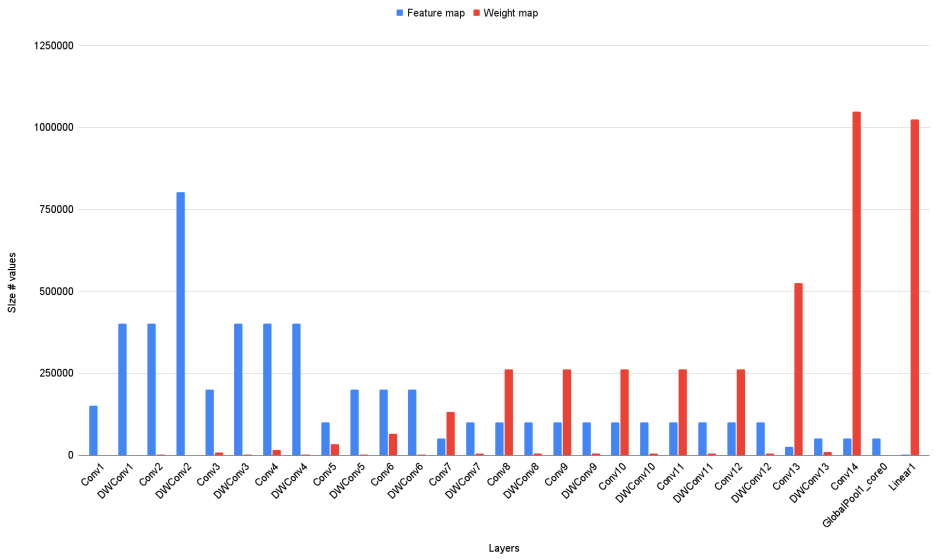


Figure 4.4: $Mobilenet$ size feature maps and weight maps layer by layer.

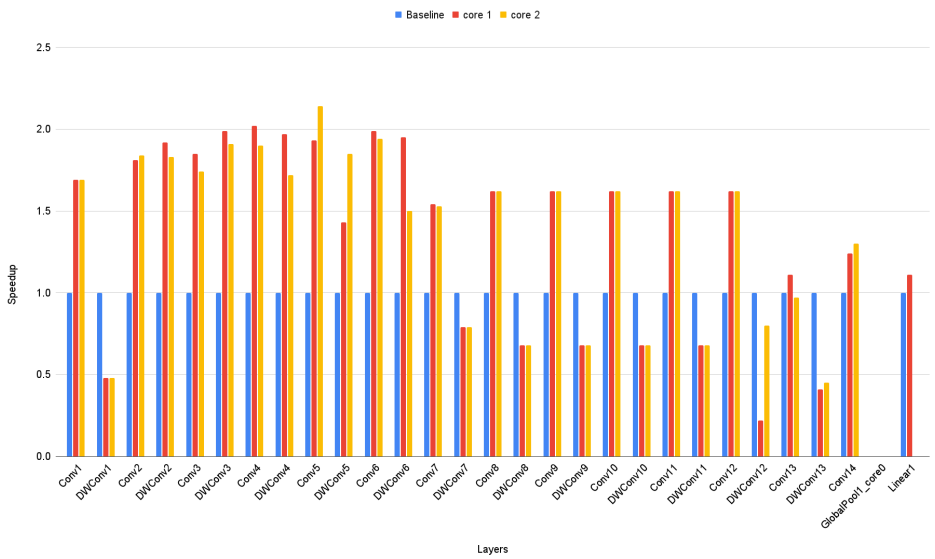
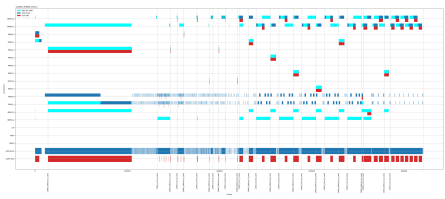
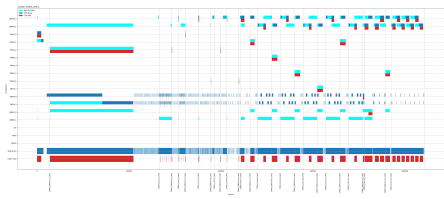


Figure 4.5: $Mobilenet$ layer by layer speedup.



(a) $Mobilenet$ inference profiler core 1.



(b) $Mobilenet$ inference profiler core 2.

Figure 4.6: $Mobilenet$ inference profiler for 2 cores

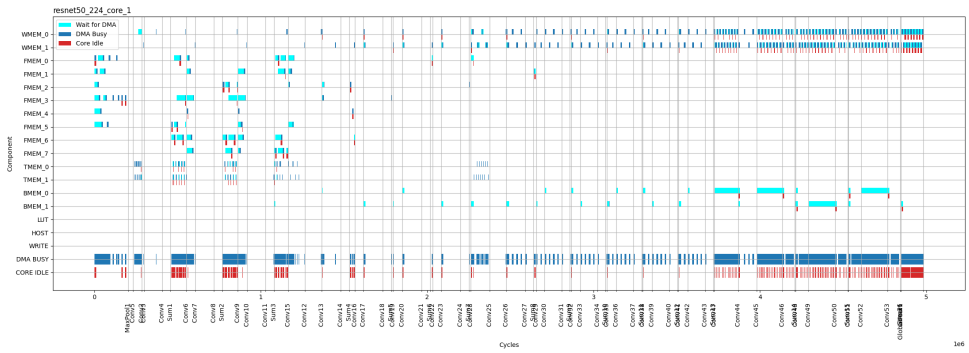


Figure 4.7: $ResNet$ inference profiler baseline.

Finally, a profiler of $Mobilenet$ divided into cores can be seen in Figure 4.6. In this profiler, it can be seen the previously mentioned error of the simulator in the second layer. Also, it can be seen how the WMEM becomes busier and busier, creating a bottleneck in the processor.

4.3.2 ResNet50 Network Analysis

$ResNet$ has been chosen to contrast with $Mobilenet$ since this one does not fit all the feature maps in the on-chip, so the division's performance is much more outstanding. In Figure 4.7, it can be seen that the layers at the beginning create load and store off-chip operation creating much overhead. Even though the core idle is not affected as much as it could seem due to the chip's pipelining method. Another characteristic of $resenet$ is the shortcuts where the feature map must be flushed and loaded entirely inside the on-chip memory.

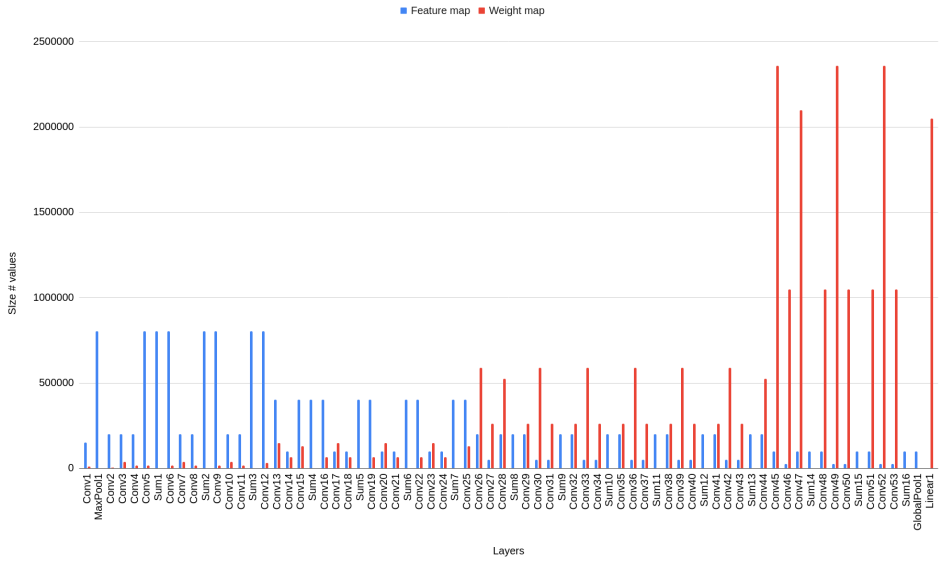


Figure 4.8: $ResNet$ size feature maps and weight maps layer by layer.

In the layer-by-layer characteristics, as it can be seen in Figure 4.8 and similar to $Mobilenet$, the layers can be grouped into three big groups:

- Feature map bigger than weight map in the case of $Mobilenet$ are layers $Conv1$ to $Conv25$.
- Feature map similar to weight map in the case of $Mobilenet$ this are layers $Conv25$ to $Conv44$.
- Feature map smaller than weight map in the case of $Mobilenet$ are layers $Conv44$ to $Linear1$.

Because there are more layers in group 3, it will decrease some of the improvement of performance attained at the beginning thanks to the reduced load and flush into the off-chip. This previous statement can be reflected in Figure 4.9, where there is a performance degradation the more the networks advances through the layers. However, the improvements thanks to that all the memory fits on-chip it achieves

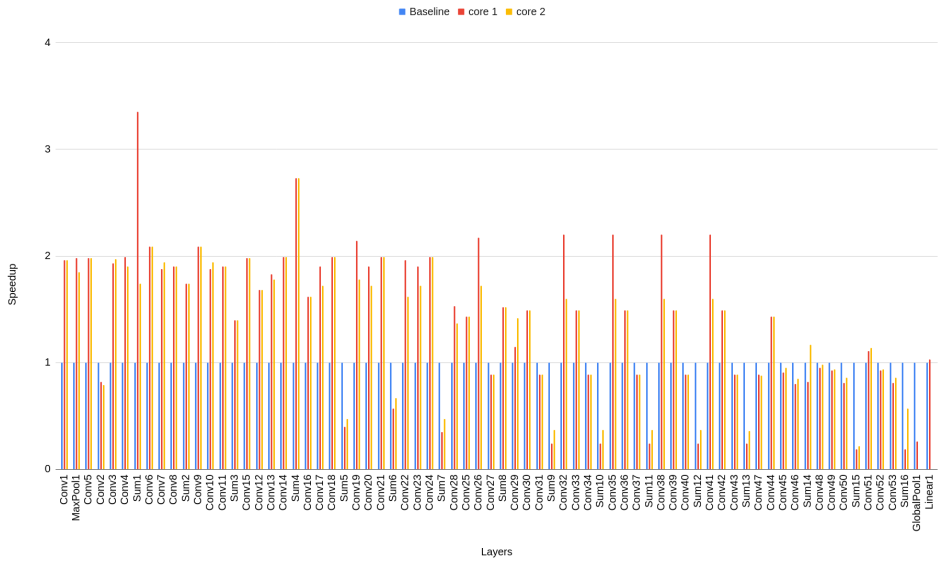


Figure 4.9: $Mobilenet$ layer by layer speedup.

better performance on group 1 than $Mobilenet$

4.3.3 Results on different networks

Now with some knowledge of some networks, the following results of Table 4.2 correspond to the speed improvement of the inference time. In that table, it can be seen interesting results ranging from an improvement of $2\times$ faster in the case of $wide\_resnet101$ to a speed reduction of $0.36\times$ in the $mobilenet\_v3\_small\_minimal$. The optimized results could only be obtained for some of the cores due to some simulator errors, so only naive results could be thoroughly analyzed on MIDAP.

The worst case is that the network size has little margin to improve the inference time. In MIDAP, like in other architectures, the small the network, the more possibilities that the whole network fits on-chip memory, and layer fusion occurs. Layer fusion in the case of MIDAP means that the data on-chip memory is reused in the

next layer, and there is no need to transfer it to the off-chip memory. This layer fusion case makes the BBGraP overhead due to the transference of data between the cores.

However, the other case can also be seen that the bigger the network layer, the overhead is less significant compared to the improvement that the division of the layers creates. Also, notice that the more layers a model has, the overhead not to be able to divide the FC layers, as explained in the previous sections, are also diminished.

These two patterns can be seen by groups being the worst cases, the $mobilenet\_v3$ and $mobilenet\_v2$ groups with an average of $0.55\times$, $0.73\times$ and $0.91\times$ for two, three and four cores respectively. And the best case on the $ResNet$ and $wide\_resnet$ groups with and average of $1.38\times$, $1.64\times$ and $1.84\times$ for two, three and four cores respectively. These results were obtained using the naive graph division and no optimization, so as will be seen in the next section, there is still much margin for improvement.

4.3.4 Comparative of using operators optimization

MIDAP simulator, since it is in development, only a few types of graphs are available, so the only ones that could be tested to check the performance of the operator's optimization are explained in Section 3.4. The graphs tested on MIDAP with optimization are $resnet50$, $resnet101$, and $resnet152$ for two cores. The results for these graphs are shown in Figure 4.10.

The improvement is relatively significant in all the cases. Even in some cases, these improvement is greater than three cores without optimization (Table 4.2). Since it could not be tested for a higher number of cores, it cannot be assumed it is linear if the number of cores is increased. However, following the previous results without optimization should be the case since they are linear.

Networks	Baseline	2 cores	3 cores	4 cores
resnet50	1	1.32	1.57	1.8
resnet101	1	1.28	1.53	1.72
resnet152	1	1.29	1.55	1.76
mobilenet	1	1.15	1.44	1.7
mobilenet_v2	1	0.62	0.84	1.07
mobilenet_v3_small	1	0.41	0.54	0.67
mobilenet_v3_small_0.75	1	0.45	0.59	0.73
mobilenet_v3_small_minimal	1	0.36	0.48	0.61
mobilenet_v3_large	1	0.57	0.76	0.94
mobilenet_v3_large_0.75	1	0.51	0.68	0.86
mobilenet_v3_large_minimal	1	0.53	0.72	0.91
mobilenet_v3_edgetpu	1	0.94	1.19	1.46
mobilenet_v3_edgetpu_0.75	1	0.59	0.77	0.98
wide_resnet101	1	1.55	1.83	2
wide_resnet50	1	1.47	1.73	1.93

Table 4.2: Speed up results for different networks and different numbers of cores on MIDAP for a naive division

4.4 AWS cluster results

In this section, differently from MIDAP, we evaluate BBGraP automatic search on different networks. To do the evaluation, an AWS cluster was used. Each cluster instance uses 8vCPUs from a custom Intel Cascade Lake, 1 NVIDIA GPU, 32GiB RAM, and a network bandwidth of up to 25Gbps. To utilize these instances, NVIDIA cuDNN was used for the GPU, and OpenMPI was used for the data transfers. The current underdevelopment simulator using OpenMPI leads to unexpected overheads due to bringing required tensors from the GPU VRAM to the host's DRAM to create the tensor and finally send it back to the GPU. This overhead will be hidden in future versions by overlapping it with inference computations using NVIDIA NCCL, leading to better inference time overall. Also, it is currently only able to divide by width, so it has the same problem on the last layers as MIDAP.

For the software side, the 2 available networks will be tested: those $VGG16$ [18]

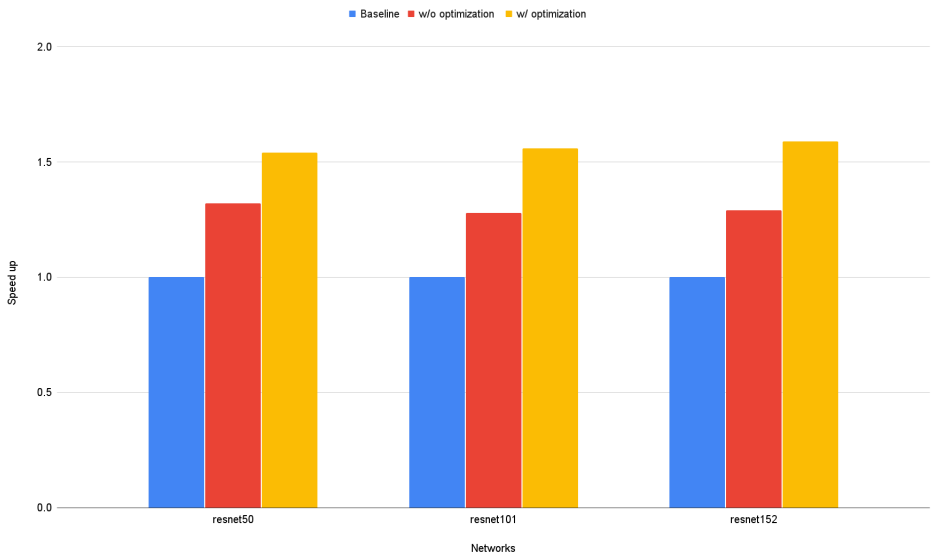


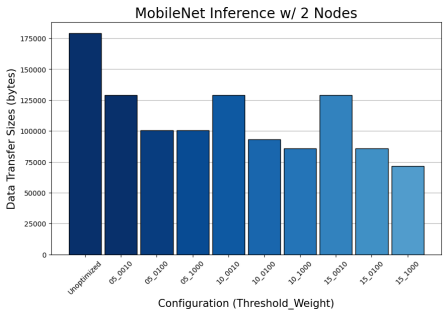
Figure 4.10: $ResNet$ comparative with and without optimization for 2 cores.

and $Mobilenet$. $VGG16$ is a network shorter than $Mobilenet$, but the feature maps are much bigger, so it should be more difficult to transfer them into the GPU. On the other hand, we have $Mobilenet$, which, as we evaluated on the MIDAP results, is a small network that, for a GPU, is not a problem to process the inference.

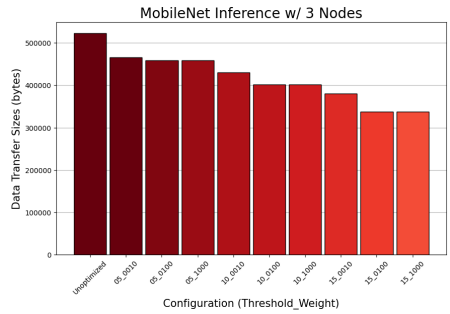
We used Pytorch in 1 node to compare the results to see if multiple nodes improve performance. We did two kinds of tests offloading the data and loading the data in every layer, and another test was not doing that.

4.4.1 Mobilenet evaluation

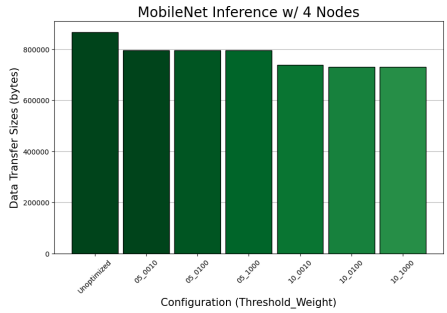
As discussed in Section , the current model is naive and does not have automatic feedback. The weights are manually selected by looking at the bottlenecks that happened during naive inference or the already known bottlenecks, like in the data transfer case. The overhead of the data transfer due to the OpenMPI is the biggest. The data transfer will be the parameter to modify and analyze in our model.



(a) Inference 2 nodes



(b) Inference 3 nodes



(c) Inference 4 nodes

Figure 4.11: How does the weight of the data transfer and threshold pruning value change the data transfer

In Figure 4.11, it can be seen that how the weight for the data transfer has a more significant impact than the threshold. These results are because the threshold only increases the possibility of finding a better solution. However, the weight for the data transfer is directly involved in the optimizing problem penalizing the higher data transfer. From the nonoptimized divided graph, the improvement to the best case is $2.33\times$, $1.42\times$, and $1.14\times$ fewer bytes transferred. The decrease in performance when the number of nodes increases is because the more nodes, the more possibilities have synchronization between them.

For $Mobilenet$, since all the data fits in the GPU, the results could be better. The more batches it has, the better results, but more is needed to show an improvement to Pytorch. In both cases, layer offloading and loading in Figure 4.12 and not doing that in Figure 4.13. The test offloading and loading of every layer has better performance

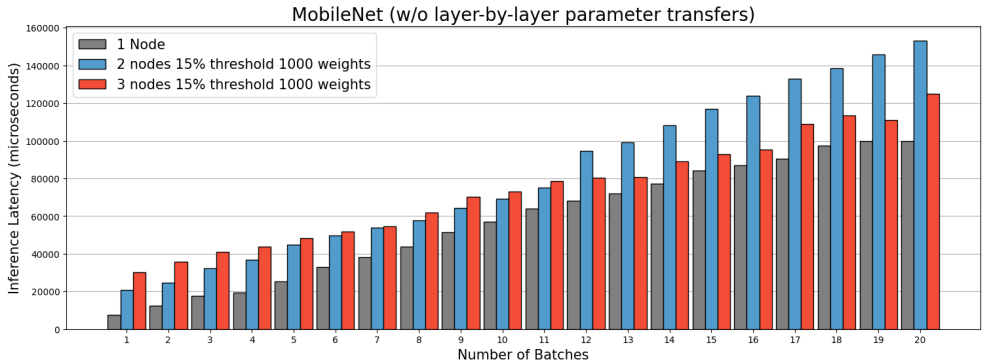


Figure 4.12: $Mobilenet$ inference without layer-by-layer offloading.

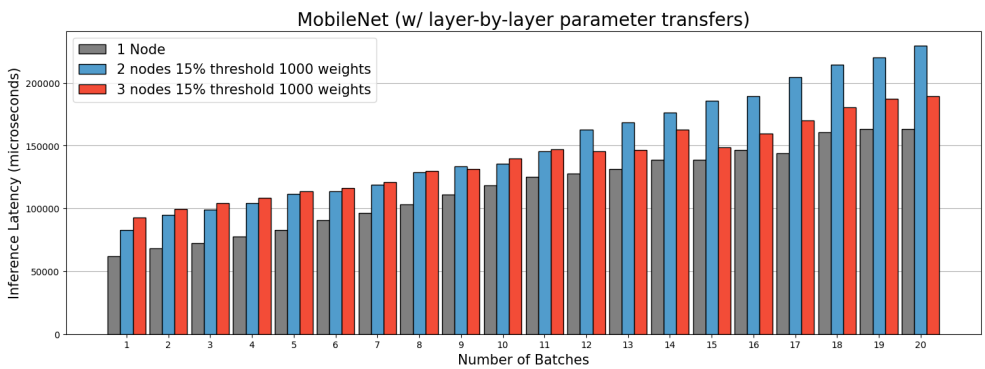
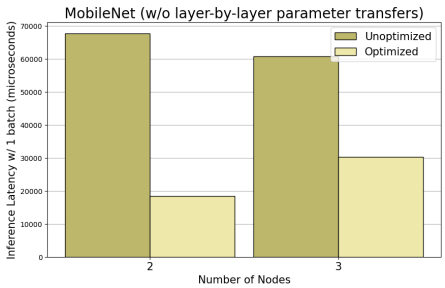


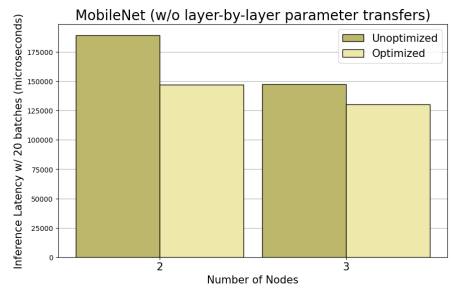
Figure 4.13: $Mobilenet$ inference with layer-by-layer offloading.

than not using this technique, but the possible overhead of the OpenMPI eclipses the improvement of the division. Another factor is the time it takes to create the automatic graph for 4 nodes; It could not be tested. However, looking at the results of three nodes, if the improvement is linear, it could have better results than Pytorch.

Finally, it will be compared to the improvement of the best graph with manually selected weights compared to the naive approach in Figure 4.14. Even though it might not be the best graph possible due to the optimization problem being too general, the inference performance is better. With the best case in two nodes, an improvement of $3.58\times$ is faster. The increase in latency with the number of nodes is due to the high data transfer in the three nodes that, as we increase the number of batches, is hidden



(a) 1 batch.



(b) 20 batches.

Figure 4.14: $Mobilenet$ best and worst case.

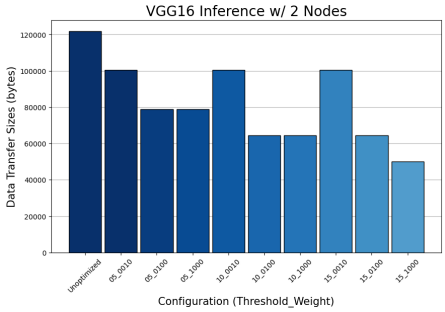
by the number of calculations the GPU has to do.

4.4.2 VGG16 evaluation

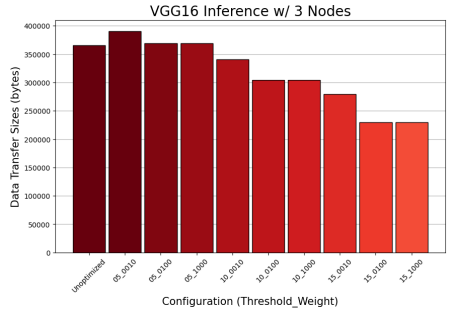
Like the previous evaluation, we will analyze the total number of bytes transferred. The results are similar to $Mobilenet$, but there is one curious case in three nodes. In the case of three nodes with a 5% threshold and the lowest value for the data transfer weight, the data transfer is higher than the nonoptimized case. These results could be due to the lack of options for choosing the partitions and because the data transfer needs to be penalized more.

In $VGG16$, similar to $Mobilenet$ at the beginning, the performance is worse due to the data transfer overhead. However, unlike in the previous case, here we can see an improvement after 5 batches. This improvement is because the feature maps of $VGG16$ are much more significant. We can see a considerable improvement on 20 batches of four nodes in Figure 4.16 with an improvement of $1.6\times$. In this case, differently from the $Mobilenet$ as it can be seen in Figure 4.17, the layer offloading and loading method does not show better results than Python but worse than without offloading and loading every layer.

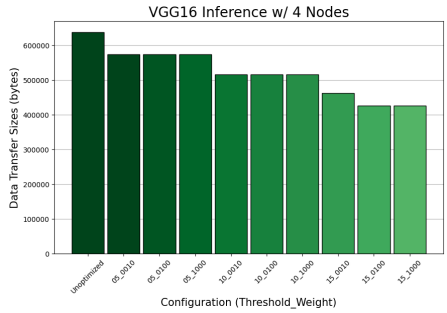
Finally, we will analyze how the nonoptimized graph improves the automatic search on 1 and 20 batches. From Figure 4.18, on only 1 batch, the improvement is



(a) Inference 2 nodes



(b) Inference 3 nodes



(c) Inference 4 nodes

Figure 4.15: How does the weight of the data transfer and threshold pruning value change the data transfer

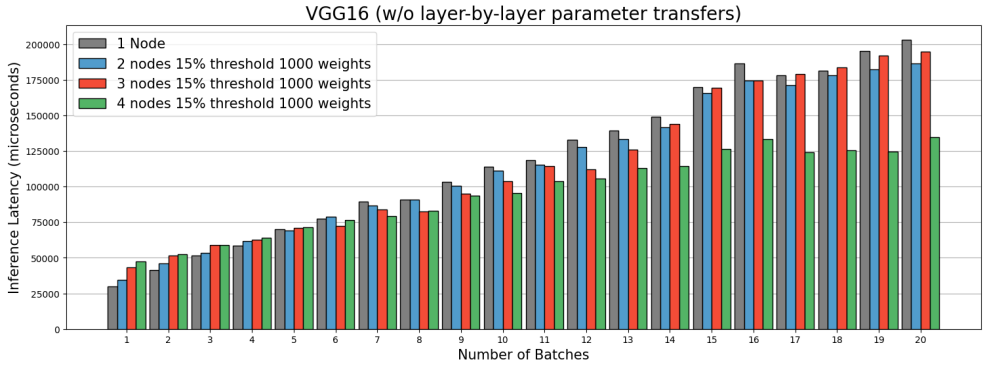


Figure 4.16: $VGG16$ inference without layer-by-layer offloading.

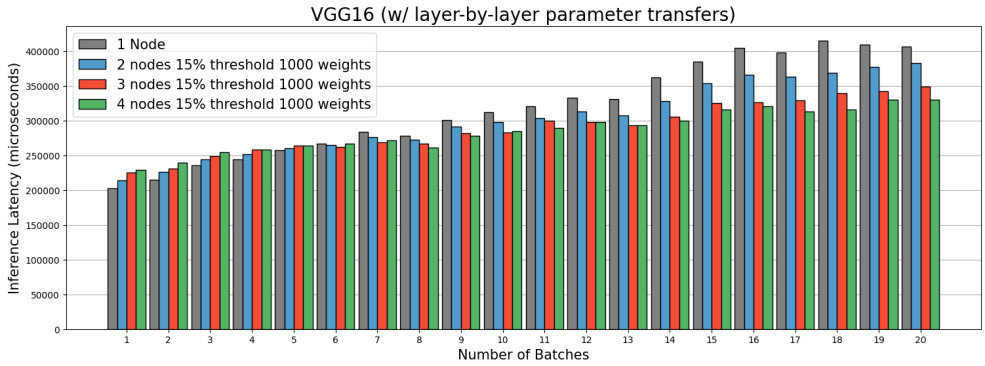
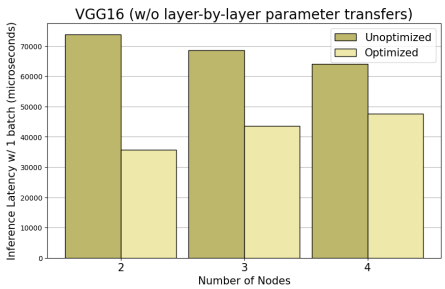
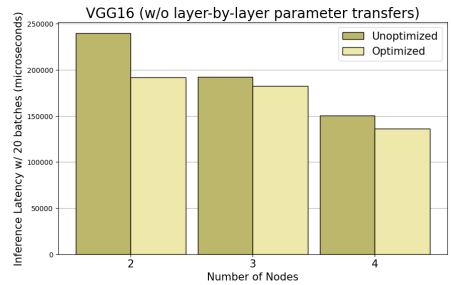


Figure 4.17: $VGG16$ inference with layer-by-layer offloading.



(a) 1 batch.



(b) 20 batches.

Figure 4.18: $VGG16$ best and worst ca.se

more remarkable since the overhead of the data transfer is not eclipsed by the total number of operations as it is in the 20 batches.

Chapter 5

Conclusion

In this thesis, we propose BBGraP, a device- and model-agnostic framework. The framework produces an efficient distribution plan for DNN inference and an automatic search to exploit the best parallelization. We described how BBGraP parses the input into its graph and partitions according to the hardware configurations. We have also described optimizing BBGraP's graph by looking at the input and the output feature maps. We explained how to create procedural partitions that can store millions and millions of partitions and how the unlimited possibilities can be reduced in a lossless manner. The results of the pruning method are auspicious, reducing the total amount of possible partitions from billions and trillions to less than 100000 possibilities. Currently, the automatic search for the best distribution is still under work, reducing the number of possible partitions even more and expanding the method to multiple directions in one network. The model representing the different architecture should be improved to represent better the components and get more accurate results. But even after all of those possible improvements the current divisions shows up to $2\times$ speed up on some networks on MIDAP and $1.6\times$ on big networks in a cluster of instances.

Chapter 6

Future Work

In future work, the number of partitions done will be reduced further because even though the results are fantastic, more is needed for a more significant number of cores and multi-layer. Because the model is too general it should be improve in a near future to get more accurate graph to improve the inference time. Another work that is currently done is the multi-layer search. This multi-layer search is developed but needs better results to add them to this thesis. After the automatic search is finished, bigger and more advanced networks will be supported together with training networks in the framework. Finally, a significant contribution would be creating a module for TensorFlow or PyTorch since neither has an intra-node model distribution.

Bibliography

[1] David Ahmedt-Aristizabal, Mohammad Ali Armin, Simon Denman, Clinton Fookes, and Lars Petersson. Graph-based deep learning for medical diagnosis and analysis: past, present and future. *Sensors*, 21(14):4758, 2021.

[2] Amazon. Amazon ec2 p4 instances: Highest performance for ml training and hpc applications in the cloud, 2020.

[3] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. Yolov4: Optimal speed and accuracy of object detection. *arXiv preprint arXiv:2004.10934*, 2020.

[4] Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic. Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation computer systems*, 25(6):599–616, 2009.

[5] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity, 2021.

[6] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[7] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[8] Donghyun Kang, Jintaek Kang, Hyungdal Kwon, Hyunsik Park, and Soonhoi Ha. A novel convolutional neural network accelerator that enables fully-pipelined execution of layers. In *2019 IEEE 37th International Conference on Computer Design (ICCD)*, pages 698–701. IEEE, 2019.

[9] Duseok Kang, Donghyun Kang, and Soonhoi Ha. Multi-bank on-chip memory management techniques for cnn accelerators. *IEEE Transactions on Computers*, 71(5):1181–1193, 2021.

[10] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.

[11] Wookey Lee, Jessica Jiwon Seong, Busra Ozlu, Bong Sup Shim, Azizbek Marakhimov, and Suan Lee. Biosignal sensors and deep learning-based speech recognition: A review. *Sensors*, 21(4):1399, 2021.

[12] Jiachen Mao, Xiang Chen, Kent W Nixon, Christopher Krieger, and Yiran Chen. Modnn: Local distributed mobile computing system for deep neural network. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 1396–1401. IEEE, 2017.

[13] Svetlana Minakova, Erqian Tang, and Todor Stefanov. Combining task-and data-level parallelism for high-throughput cnn inference on embedded cpus-

gpus mpsocs. In *International Conference on Embedded Computer Systems*, pages 18–35. Springer, 2020.

[14] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. Pipedream: generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 1–15, 2019.

[15] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021.

[16] Joseph Redmon and Ali Farhadi. Yolo9000: better, faster, stronger. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7263–7271, 2017.

[17] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[18] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[19] Rafael Stahl, Alexander Hoffman, Daniel Mueller-Gritschneder, Andreas Gerstlauer, and Ulf Schlichtmann. Deeperthings: fully distributed cnn inference on

resource-constrained edge devices. *International Journal of Parallel Programming*, 49(4):600–624, 2021.

[20] Fei Sun, Jun Liu, Jian Wu, Changhua Pei, Xiao Lin, Wenwu Ou, and Peng Jiang. Bert4rec: Sequential recommendation with bidirectional encoder representations from transformer. In *Proceedings of the 28th ACM international conference on information and knowledge management*, pages 1441–1450, 2019.

[21] Zhuoran Zhao, Kamyar Mirzazad Barijough, and Andreas Gerstlauer. Deepthings: Distributed adaptive deep learning inference on resource-constrained iot edge clusters. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2348–2359, 2018.

요약

최근에 다양한 도메인에서 개발 및 사용하는 심층 신경망들의 크기는 점점 더 커지는 추세이다. 이로 인해 딥 러닝 애플리케이션을 개발하려는 기업과 개인은 고가의 하드웨어 설비를 구입하거나 클라우드 공급자로부터 하이엔드 머신을 임대해야 하며, 이는 사용자들에게 큰 부담으로 다가온다. 수십 억 또는 수조 개의 매개 변수를 가지는 심층 신경망을 실행할 수 있는 노드들을 사용할 때 발생하는 높은 비용을 피하는 대안 중 하나는 더 저렴하지만 느린 노드들을 동시에 사용해 작업량을 분산하는 것이다. 작업량 분산과 애플리케이션의 지연 시간 요구 사항들을 동시에 충족하면서 각 노드의 메모리 및 계산 요구 사항에 따라 작업량을 조정하기 위해 여러 병렬화 전략들이 제안됐지만, 이러한 방법들을 사용자가 직접 사용하기 위해선 분산화 전략과 심층 신경망에 대한 상당한 지식을 보유해야만 한다. 이러한 문제점을 해결하고자 본 논문에서는 하드웨어 설비 및 심층 신경망 모델에 구애받지 않고 손쉽게 딥 러닝 추론을 위한 효율적인 병렬화 계획을 생성하는 블랙박스 그래프 분할기인 BBGraP를 제안한다. BBGraP을 통해 주어진 각 노드의 메모리 및 계산 제약 조건에 따라 효율적인 워크로드 분할을 생성하며, 사용자들이 원하는 다양한 유형의 심층 신경망들을 실행할 수 있다. BBGraP에서 분할 방식을 고안할 때 사용하는 그래프 최적화 도구는 중복 작업, 데이터 전송 및 동기화 지점을 제거하여 전송되는 데이터 양을 줄여 워크로드의 지연 시간을 개선하고, 그 후에 자동 검색 방식이 지정된 설정에 따라 가능한 최적의 파티션을 찾게 된다. 이러한 방식들을 통해 여러 노드를 포함한 클러스터와 다중 코어 FPGA에서 최고 2배의 성능 향상을 보여주는걸 확인할 수 있었다.

주요어: 심층 신경망, 병렬화, 컴파일러, 자동 검색기, 블랙박스, 그래프 최적화

학번: 2020-27033

Acknowledgements

I want to thank all the people that have been around me these two years, helping and supporting me. First of all, I want to dedicate this thesis to both of my beloved parents that always supported me since I was born and always were there to help even in the distance. I would like to thank my friends and the rest of my family in Spain that helped me during these two tough years with calls, messages, or video calls since I couldn't go there. I want to thank my advisor Bernhard Egger for allowing me to do my research in his lab and also for the advice through these years. Also, I'm thankful to my lab mates for the long hours in the lab, the exchange of ideas and also help. Finally, I want to thank professor 이재진 and professor 김진수 for the useful feedback for this thesis.

Gràcies per tot pare i mare per sempre estar allí i ajudar-me.
Sempre et recordaré pare.