UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

RAFAEL FÃO DE MOURA

Software-only Computation Reuse Techniques for Energy Efficient CNNs

Thesis presented in partial fulfillment
of the requirements for the degree of
Master of Computer Science

Advisor: Prof. Dr. Luigi Carro

Porto Alegre
julho 2019

CIP — CATALOGING-IN-PUBLICATION

Fão de Moura, Rafael

Software-only Computation Reuse Techniques for Energy Efficient CNNs / Rafael Fão de Moura. – Porto Alegre: PPGC da UFRGS, 2019.

67 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2019. Advisor: Luigi Carro.

1. Convolutional neural networks. 2. Computation reuse. I. Carro, Luigi. II. Título.

“Not everything that can be counted counts
and not everything that counts can be counted.”

— Albert Einstein

ACKNOWLEDGMENTS

I want to thank and dedicate this thesis to every person who has been involved in my academic life. First, I would want to give special thanks to Professor Luigi Carro for his guidance and words of motivation and inspiration. I want to convey my gratitude to João Paulo, Paulo, and Larissa for the discussions and cooperation in many works. Finally, I would like to thank all my beloved relatives and friends, specially Denise, who have been so supportive along the way of my academic journey.

ABSTRACT

In the past years, several efforts in algorithm and architectural research were put together to enable large-scale use of CNNs as we know today. Thus far, most of these achievements have been based on improving convolutions by chasing the parallel execution of MAC operations through the replication of floating-point units. However, these solutions fall far short of what is allowed from the energy budget when it comes to embedded systems running these NN models. Given specific image characteristics, such as recurrent input patterns, we propose an algorithmic changing for performing CNN inferences by employing a computation reuse technique instead of the original implementation. Based on statistical analysis, we address computation reuse at three granularity levels: convolution kernel-level and grid-level through employing lookup tables in place of the original convolutions, and frame-level by replacing entire frame computations with a movement prediction algorithm. Experimental results show that it is possible to achieve energy savings up to 27.5×, while reducing the inference time to 116× of the baseline, with an accuracy loss of 13%.

Keywords: Convolutional neural networks. computation reuse.

Técnicas de reuso de computação em software para CNNs energeticamente eficientes

RESUMO

Nos últimos anos, pesquisas em melhorias nas áreas de algoritmos e arquiteturas computacionais foram postas lado-a-lado de modo a permitir o uso em larga-escala de CNNs. Desde então, a maior parte destas melhorias têm sido baseadas na aceleração de convoluções através da execução paralela de operações MAC, utilizando a replicação de unidades de ponto-flutuante. No entanto, essas soluções ficam muito aquém do que é permitido em termos de consumo energético quando se trata de sistemas embarcados executando NNs. Considerando características específicas de imagens, tais como repetições de padrões de entrada, neste trabalho, nós apresentamos uma mudança algorítmica no modo como CNNs realizam inferências, empregando uma técnica de reuso de computação no lugar da implementação original. Com base em análises estatísticas, nós abordamos o reuso de computação em três granularidades: ao nível de convolução e ao nível de conjunto de convoluções, realizando consultas em tabelas ao invés das convoluções originais, e ao nível de frame através da substituição da computação original de um frame inteiro por um algoritmo de predição de movimento. Nossos resultados mostram que é possível obter níveis de economia de energia em até 27,5×, e reduzir o tempo de inferência por um fator de 116× em relação à versão original, com uma perda de precisão de 13%.

Palavras-chave: redes neurais convolutioncais, reuso.

LIST OF ABBREVIATIONS AND ACRONYMS

AI	Artificial Inteligence
CNN	Convolutional Neural Network
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
DRAM	Dynamic Random Access Memory
DVFS	Dynamic Voltage and Frequency Scaling
FPGA	Field-Programmable Gate Array
FPS	Frames Per Second
FPU	Floating-Point Unit
FU	Functional Unit
GPP	General Purpose Processor
GPS	Global Positioning System
GPU	Graphics Processing Unit
IoT	Internet of Things
LLC	Last-Level Cache
MAC	Multiply-accumulate
MLP	Multilayer Perceptron
MV	Motion Vector
NN	Neural Network
ReLU	Rectified Linear Units
SAD	Sum of Absolute Difference
SIMD	Single Instruction Multiple Data
SRAM	Static Random Access Memory
TSS	Three Step Search

LIST OF FIGURES

Figure 1.1 FPS and energy per frame running CNNs in several architectures 13

Figure 2.1 A generic model of CNN 18
Figure 2.2 Convolution layer 19
Figure 2.3 Redundancy ratios running several CNNs over different datasets and batch sizes 21

Figure 4.1 Execution flow for the proposed profiler tool 29
Figure 4.2 Execution flow for the convolution kernel-level reuse 30
Figure 4.3 Table size (Megabytes) to keep in storage convolution values for a batch of 10,000 inputs 30
Figure 4.4 Histogram of convolution outputs for three CNN models with different datasets 32
Figure 4.5 Histogram of output values before and after clustering 34
Figure 4.6 Mapping Functions and regular convolution execution time comparison 35
Figure 4.7 Percentage of repeated 2x2 submatrices throughout the layers of different CNN models and datasets 36
Figure 4.8 Iterations of the algorithm using computation reuse of adjacent activations 39
Figure 4.9 CNN and motion prediction comparison. 40
Figure 4.10 CNN execution histogram distribution 41
Figure 4.11 Complete execution flow. 42

Figure 5.1 Table sizes and accuracy for the convolution kernel-size reuse technique 46
Figure 5.2 Average inference time comparison for the convolution kernel-size reuse technique 47
Figure 5.3 Energy consumption comparison for the convolution kernel-size reuse technique 48
Figure 5.4 Table sizes and accuracy for the convolution grid-size reuse technique 51
Figure 5.5 Average inference time comparison for the convolution grid-size reuse technique 52
Figure 5.6 Energy consumption comparison for the convolution grid-size reuse technique 54
Figure 5.7 Average frame inference time comparison over the baseline system 57
Figure 5.8 Energy gains over baseline and data transfers running three different datasets on YOLOv3-tiny. 58
Figure 5.9 Energy gains over baseline of the CAVIAR dataset using YOLOv3. 60

LIST OF TABLES

Table 3.1 Related work comparison 26

Table 5.1 System configuration. 44
Table 5.2 Grid-level reuse statistics 49
Table 5.3 YOLOv3-tiny CNN accuracy and inferred frames 55
Table 5.4 YOLOv3 variations comparison 60

Table 6.1 Comparison between this work and related work 62

CONTENTS

1 INTRODUCTION 11
1.1 Contributions 15
1.2 Organization 15
2 BACKGROUND 17
2.1 CNN basics 17
2.1.1 Input layer 17
2.1.2 Convolution layer 18
2.1.3 Activation function 19
2.1.4 Pooling functions 19
2.1.5 Fully-connected and classification layers 20
2.2 Data locality, redundancy, and computation reuse opportunities in CNN execution 20
3 RELATED WORK 23
3.1 Custom hardware accelerators for NN execution 23
3.2 Software-based techniques for efficient NN execution 24
3.3 Work comparison 25
4 PROPOSED APPROACH 27
4.1 Convolution kernel-level reuse 28
4.1.1 Performing data analysis to reduce table size 31
4.2 Grid-level reuse 35
4.3 Frame-level reuse 38
4.3.1 Execution flow 41
5 EVALUATION AND ANALYSIS 43
5.1 Convolution kernel-level reuse 44
5.1.1 Lookup table sizes and accuracy 45
5.1.2 Inference time 45
5.1.3 Energy consumption 46
5.2 Grid-level reuse 48
5.2.1 Lookup table sizes and accuracy 49
5.2.2 Inference time 50
5.2.3 Energy consumption 52
5.3 Frame-level reuse 53
5.3.1 Accuracy 55
5.3.2 Inference time 56
5.3.3 Energy consumption 57
5.3.4 Applying the technique to a deeper NN 59
6 CONCLUSION AND FUTURE WORK 61
6.1 Published Papers 62
REFERENCES 64

1 INTRODUCTION

Convolutional Neural Networks (CNNs) comprise a class of supervised learning algorithms that are specialized in recognizing patterns and properties from the inputs that are fed to them. The behavior of a CNN is inspired by the brain's basic operation in which the algorithm replicates the same structure and connections present in organic nervous systems. The first attempts to build Neural Networks (NNs) employed a neuron model called Perceptron (ROSENBLATT, 1958). Then, this model was used to create a more robust representation composed of a few neurons and layers, which are known as Multilayer Perceptron (MLP). Although MLPs present limited levels of accuracy and applicability, they have been used for a quite long time. Recently, they have evolved towards deeper networks such as CNNs.

When LeCun et al. (1990) presented a groundbreaking CNN model, which introduced the multi-layer CNN concept relying on the backpropagation algorithm, it was possible to classify handwritten digits with a satisfactory level of accuracy. This new model allowed NNs to learn the position and scale-invariant structures in the data, which is essential for many types of prediction problem involving image data as input. Ever since, CNNs have become a standard machine learning solution for several modern application realms, such as image classification, object detection, speech processing, and other problems involving non-image data (LIU et al., 2017; REDMON; FARHADI, 2017; LECUN et al., 2015).

Among all types of layers that compose a CNN layout, the more significant time slice in the CNN inference process is spent on convolution layers (JIAO et al., 2018). At their core, convolution layers perform the extraction of features through convolutions, which can be seen as multidimensional dot-product operations, thus relying massively on Multiply-accumulate (MAC) calculations. For instance, up to 92% and 86% of the execution time in a Graphics Processing Unit (GPU) and a General Purpose Processor (GPP), respectively, is spent on convolution layers during the forward process for the CNN proposed by Krizhevsky, Sutskever and Hinton (2012). Hence, the industry's efforts to accelerate CNN execution have been concentrated on optimizing hardware design for convolution operations, ranging from conventional architectures, such as GPPs (SODANI, 2015) and GPUs (CHOQUETTE; GIROUX; FOLEY, 2018), to new custom-accelerator based on high-performance Single Instruction Multiple Data (SIMD) and MAC engines (CHEN et al., 2014b).

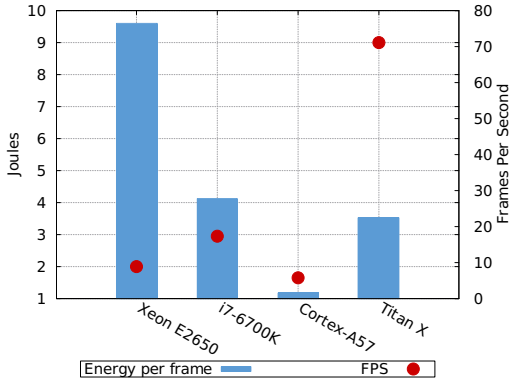
With the advent of the Internet of Things (IoT) era, there is a trend to migrate the execution of such Artificial Intelligence (AI) algorithms, previously done in specialized data centers and clusters, to local devices, using low-cost Central Processing Units (CPUs), as already seen in embedded systems (ZOU et al., 2019). The increasing demand for prediction and recognition tasks in smartphones and self-driving vehicles are two good examples to illustrate the shift that embedded devices are going through nowadays. For the autonomous vehicles environment, many different sensors, such as cameras, Global Positioning System (GPS), accelerometers, gyroscopes, and ultrasonic devices are presented to employ different functionalities such as pedestrian detection, collision avoidance, gear control, localization, and self-parking (AZMAT; SCHUHMAYER, 2015). Smartphones also rely on several sensors to gather information and to process it on specialized NN algorithms, such as face recognition, speech processing, and fingerprint authentication.

Nevertheless, the execution of CNNs still faces challenges related to chip area, energy consumption, and time requirements on embedded systems, especially regarding lightweight devices with limited resources and power budgets or running on tight latency constraints. Despite dedicated hardware accelerators have advanced the state-of-the-art on CNN execution, their practical implementation is still costly, as they either insert high production costs or break the software design life cycle. On the other hand, traditional architectures, such as GPPs and GPUs, do not hurt programmability and provide performance-efficiency outcomes along with the massive replication of Functional Units (FUs) and processor cores by chasing the parallel execution of MAC operations, which in turn increases area and power dissipation on chips.

To illustrate the performance versus energy-efficiency relationship, Figure 1.1 presents the Frames Per Second (FPS) ratio and the energy costs to perform a frame inference running the YOLO9000 CNN (REDMON; FARHADI, 2017) on two setups of GPPs, a GPU, and a typical embedded processor. It is possible to notice that the GPU presents the best FPS ratio, while it requires a considerable amount of energy to process a frame. On the other hand, the embedded processor presents the best energy saving. However, this comprises the worst performance result.

In an ideal scenario, it is always desirable to achieve both the performance inherit from high-performance systems like GPUs and the low-energy cost brought by embedded processors. One possible approach to tackle this issue is through the replacement of the original algorithm by a more efficient version. Several works have introduced advances

Figure 1.1: FPS and energy per frame running CNNs in several architectures



Source: Author

in new ways to replace the original CNN execution with software-only changes (LIU et al., 2018). In general, the most common approach taken in an algorithmic change for the CNN realm is through the exploitation of mathematical or statistical characteristics present in the execution of these CNNs.

An essential but yet not well-explored characteristic in CNNs acceleration field is the fact that the layout of CNNs presents data compression behavior along with the different set of layers that compose the network. As the input data is passing through a CNN, the temporary data generated is transformed and packed into smaller matrix representations according to the parameters that define the network topology, such as the number and size of filters, stride, padding, and downsampling elements. For instance, to perform inference over an input image in YOLOv3-tiny CNN (REDMON; FARHADI, 2018), nearly 10 GB of temporary data is generated and processed between different layers. However, the input is a 416x416 RGB image, and the output is only a classifier vector output represented into 320 Bytes. These measurements give us a glance to the opportunities for compression, which comes from the high level of data redundancy, to reduce the temporary data produced throughout the forward process.

In the meantime, different domains where image recognition is applied commonly present high levels of data redundancy. This property, when combined with the CNN data compression behavior, enables computation reuse within a single input image or consecutive frames since pixels in the neighboring region tend to exhibit very similar values, and an image streaming over the same scenario generally changes by subtle or often imperceptible degrees. Moreover, these replicated groups of pixels are forwarded to the next CNN layers, with data patterns being repeated throughout the whole network execution. Recently, computation reuse has been proposed as an alternative for exploiting input recurrence in CNN. Some straightforward implementations of this idea replace a floating-

point multiplication by a table lookup, commonly implemented as an associative memory (RAZLIGHI et al., 2017; JIAO et al., 2018; MOCERINO; TENACE; CALIMERA, 2019). Nonetheless, the fine-grain approach is incapable of providing a significant factor of energy and time savings because it does not tackle a more critical issue: reduce the number of tasks, whether they can be floating-point or lookup operations.

Further, exploiting frame similarities over inputs is one fundamental idea behind video and image compression algorithms. They are based on the idea that statistically, the set of observed inputs tends to present only a small portion of the all possible combinations, thus generating an even more reduced set of outputs. These algorithms can take advantage of the data recurrence by using motion estimation techniques, which work by calculating motion vectors between frames (CHEN; HUNG; FUH, 2001; ZHU et al., 2018). Thus, it is possible to keep information representing an object moving through time and predict upcoming frame information.

Supported by that compression behavior present in CNNs execution and input patterns recurrence, we propose an algorithmic changing to the CNNs execution employing a closer look at wider granularities of data reuse. In this work, we exploit computation reuse opportunities in CNNs execution due to input repetition by addressing three different levels of reuse: convolution kernel-size, grid of convolutions, and frame similarity. The first two approaches exploit the data recurrence enclosed within surrounding pixel regions in the same input image by replacing the execution of entire convolutions by lookup table accesses. Relying on video compression basics, in the third level of computation reuse exploitation, we predict object movement between frames without the need to run the full CNN for every frame.

To accomplish this, we present a preliminary analysis of training datasets and algorithmic changes for the original CNN implementation. Firstly, a statistical analysis is conducted over the intermediate data of CNN models, and a clustering mechanism reduces the amount of memory required to store the most representative entries in reuse tables. Then, the proposed technique associates every pair of input and feature map weight operands (kernel-level granularity of reuse) related to a single convolution with an activation retrieved from hash tables.

Further, we expand the granularity of computation reuse from the kernel-level to the convolution grid-level reuse based on two observations: a) the presence of recurrent background in real-world images and b) the probabilistic distribution of a pixel, where the neighboring region of a pixel can be interpreted as a normal distribution. In the grid-level

granularity, groups of adjacent activations in the feature map can be stored for further reuse, which enables us to use them in an approximate technique. By storing the most common sequences of convolution grids in the hash table, we can speculate the values of adjacent activations and check them in the future, while reducing the number of hashing operations and table lookups by up 4.5×.

Finally, we combine computation reuse and block matching techniques to exploit frame similarities in camera videos to aggressively reduce the execution time and energy consumption, using only low-cost software modifications. By implementing these techniques in the open-source *Darknet* framework (REDMON, 2013–2016), it is possible to achieve a performance speedup up to 116 times in one inference, while achieving energy gains of 27 times regarding an accuracy loss of 13% percent due to output values prediction.

1.1 Contributions

The main contributions of this work are listed below:

- The proposed techniques differ from previous state-of-the-art works since they address the computation reuse of convolution results and adjacent activation values.
- We exploit an intrinsic property of the input, which is data repetition at intra-frame and inter-frame levels to achieve an algorithmic change in the original CNN implementation.
- We present a tool that automates the generation of tables for the computation reuse technique, reduces the storage size required for these tables, and identifies patterns of recurrent activations for grid-level reuse.
- By employing a motion prediction algorithm in place of the CNN execution, we can skip the entire inference process avoiding the CNN call for every input frame.

1.2 Organization

The rest of this work is organized as follows.

- Section 2 presents a general overview of the CNN layout, execution, and introduces the concepts of data repetition found in CNN inference.

- Then, state-of-the-art endeavors to accelerate CNN are presented in Section 3.
- The proposed technique and their implications are presented in Section 4.
- Section 5 introduces the methodology, experimental setups, and results to evaluate this work.
- Finally, Section 6 presents a brief conclusion and discusses future works.

2 BACKGROUND

In this section, the basics of CNNs layout and execution are presented. Further, an analysis of the presence of data locality, data redundancy, and computation reuse opportunities in CNNs is done.

2.1 CNN basics

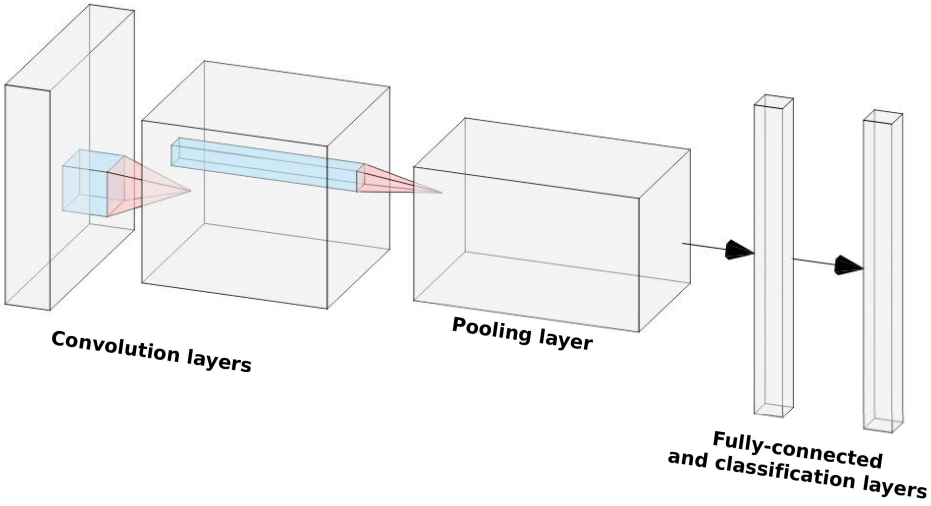
CNN is a supervised machine learning model, which has largely been employed for image classification, object detection, and other tasks. A CNN layout can be seen as a pipeline of layers with several data conditioners, such as activation functions, pooling, and normalization techniques. Figure 2.1 presents a generic architecture of a CNN. One can observe that a lot of data transformations are performed across the layers regarding any arbitrary input, helping the network to extract useful features over the data. Typically, increasing the network depth is one way to achieve higher recognition accuracy rates. For instance, state-of-the-art CNNs such as GoogLeNet (SZEGEDY et al., 2015) and YOLOv3 (REDMON; FARHADI, 2018) use, respectively, 20 and 75 convolution layers, compared to Alexnet's (KRIZHEVSKY; SUTSKEVER; HINTON, 2012) only 5 convolution layers. More detailed information about CNNs and their basic building blocks will be presented in the following sections.

2.1.1 Input layer

The input layer, also known as the data layer, is the first component present in a CNN organization. It is in charge of taking the network input (images, text, videos, etc.) and performing primary data extraction and conversions, further to feed the next elements that compose the network. The most frequent data conversion done in the input layer is the splitting of the input in *channels*. As an example, for image processing CNNs in the Darknet framework (REDMON, 2013–2016), the data layer typically splits the input images into three channels, each one representing the Red, Green, and Blue color model.

Meanwhile, it is beneficial for CNNs not to become susceptible to noise sources found in datasets. Thereby, it is desirable to have as many samples as possible to feed the network model, thus preventing *over-fitting*. The over-fitting effect happens when the

Figure 2.1: A generic model of CNN



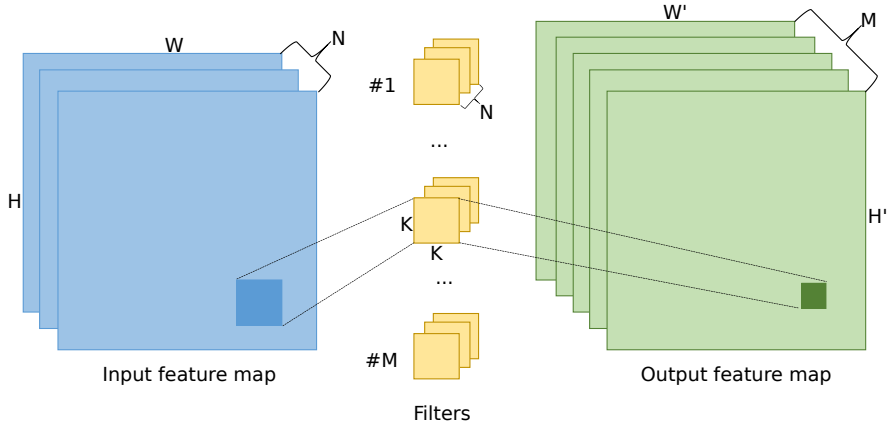
Source: Author

network model adjusts itself to generate the expected outputs given a training dataset, though the model reveals ineffective regarding new and unknown inputs. Hence, when the input batch size is not large adequately, several techniques modify the input layer to obtain more samples, relying on transformation over inputs. Some of these techniques refer to data augmentation, including image translations, shifting, mirroring, and others.

2.1.2 Convolution layer

Among all types of layers that compose the CNN layout, the most significant for the network inference process, and computationally expensive is the convolution layer. Convolution layer's essential operation is inspired from the neurons of a human brain. Figure 2.2 shows an example of a convolution layer. Each convolution layer treats as its input a set of N *channels* or N *feature maps*, each of size HxW. To generate the convolution layer outputs, or neuron outputs, the N *feature maps* are convolved with M filters or kernels of size $NxKxK$ in a multidimensional dot-product operation. The filters represent the weights that were previously obtained in the training phase using a learning algorithm such as back-propagation. For each filter of the ***M*** sets, the convolution is performed by sliding the kernel across the input feature map according to a stride value. At each overlapping of the filter over the input feature map, the values are multiplied and accumulated together, giving one value to the output feature map. Thereby, as a result of the convolution process, M output feature maps are generated with size $H'xW'$ each.

Figure 2.2: Convolution layer



Source: Author

2.1.3 Activation function

Commonly, the convolution layers are followed by the execution of activation functions. Activation functions introduce non-linearity in the data, which in turn allows the network to learn how to recognize and to extract more complex features, reaching higher accuracy levels. Although there are several implementations of these functions, the most employed in CNN domain is the Rectified Linear Units (ReLU). The ReLU rectifier's equation can be defined as the maximum value between the current input value (x), and zero ($max(0, x)$). Thereby, after the execution of a ReLU rectifier, a neuron with a ReLU output greater than zero is called activated or turned on, whereas a neuron with an activation value equals to zero is said deactivated or turned off. Recent works have observed those network models that employ ReLUs to perform inference and training operations several times faster than former network models relying on *sigmoid* and *hiperbolic*-based functions with no degradation in the accuracy.

2.1.4 Pooling functions

As the input passes through the network layers, which the typical flow is a sequence of convolution layers, each one followed by activation functions, them a pooling function is often employed. Pooling functions are applied to reduce the number of features through the pipeline of layers while retaining all the essential elements required for accurate inference. The traditional pooling implementations typically reduce the feature map sizes by performing operations such as average (Average Pooling) and obtaining the

maximum value (Max-pooling) over a grid of values, thus reducing the feature size proportionally to the input pooling grid size. Notwithstanding pooling layers capability to reduce feature map sizes without losing data representativeness, studies have pointed out that convolution layers with increased stride sizes can replace pooling without incurring a loss in the accuracy (SPRINGENBERG et al., 2014).

2.1.5 Fully-connected and classification layers

Inside each output feature map computation in a convolution layer, a single weight kernel is slid over the input feature maps. This property is known as parameter sharing, and it grants the reduction of the number of parameters (weights), as the execution time of the network. In contrast to convolution layers, fully-connected layers are composed of fewer neurons, where every single neuron is connected to all next layer neurons, thus creating an $N : N$ relationship. Fully-connected layers are commonly inserted in a CNN layout after a sequence of several convolution layers, activation functions, and pooling layers. Their primary goal is to perform high-level reasoning (ZEILER; FERGUS, 2014), which is the first step to enable the network model to generalize and reduce the input to several generalized features.

Finally, the last layer that composes a CNN architecture is an output or classification layer. Classification layers deliver multi-class ranking employing heuristics such as the *softmax* to estimate the probability of an arbitrary input to belong to a particular class. As a result of the classification layer execution, a vector classifier outcome is created, comprising all the probabilities of the current input to match in the reasonable classes that the network is capable of detect.

2.2 Data locality, redundancy, and computation reuse opportunities in CNN execution

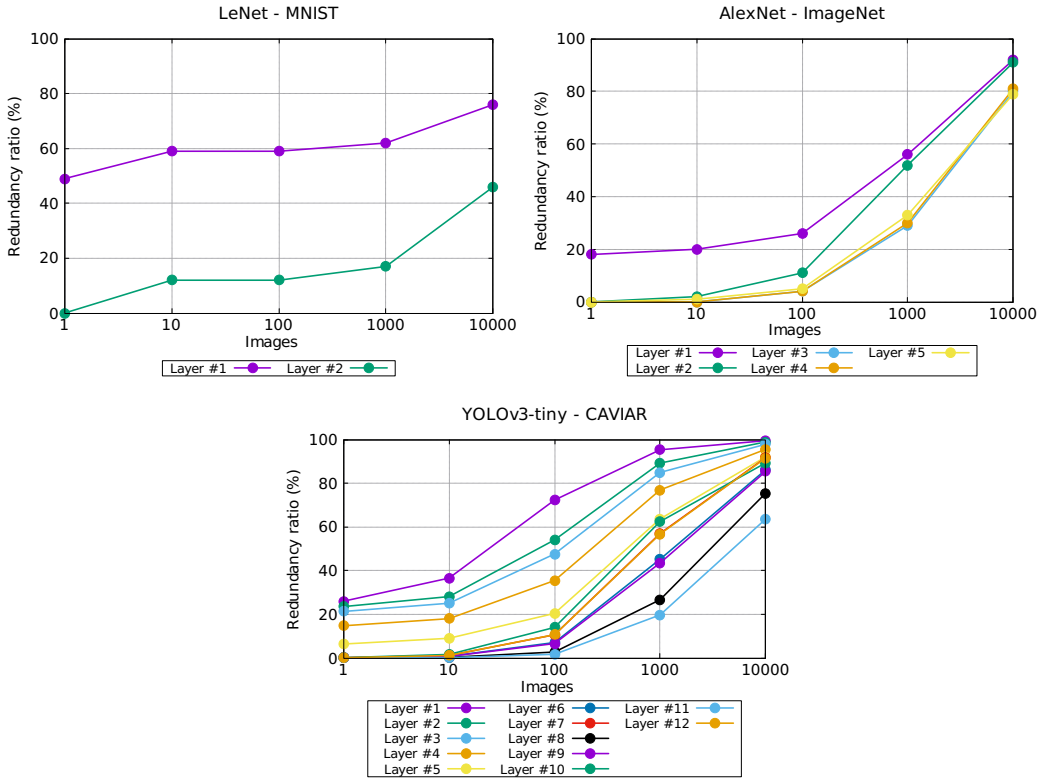
In different scenarios for CNN inference, the input images commonly expose similarities within a single sample (RIERA; ARNAU; GONZÁLEZ, 2018) and among different frames (ZHU et al., 2018). Likewise, it may expect the intermediate data generated along with the processing of convolution layers to present elevated levels of redundancy. To illustrate that redundancy levels regarding a few input datasets and CNNs layouts ex-

ecution, we have taken three distinct CNNs models and input datasets to help explain the main concepts. Figure 2.3 presents the *Redundancy Ratio* regarding inputs with convolution kernel-size granulatiry for each convolution layer, when running the datasets MNIST (DENG, 2012), IMAGENET (DENG et al., 2009), and the CAVIAR benchmark (FISHER; SANTOS-VICTOR; CROWLEY, 2004) on pre-trained CNN models for LeNet (LECUN et al., 2015), AlexNet (KRIZHEVSKY; SUTSKEVER; HINTON, 2012), and YOLOv3-tiny (REDMON; FARHADI, 2018), respectively. The *Redundancy Ratio* is a metric built to illustrate the input patterns recurrence observed during batch execution. This metric is calculated and defined by the following equation:

$$Redundancy_{ratio} = \left(1 - \frac{\#DifferentInputPatterns}{\#TotalInputPatterns}\right) \times 100 \qquad (2.1)$$

Where *#Different Input Patterns* refers to how many different inputs were observed during the execution, and *#Total Input Patterns* means all the inputs gathered during the batch size execution. Hence, a *Redundancy Ratio* of zero percent means that all the input patterns observed were different from each other, and a *Redundancy Ratio* of 50% means that, on average, each input appeared twice in the batch execution.

Figure 2.3: Redundancy ratios running several CNNs over different datasets and batch sizes



Source: Author

Regarding only one input sample, the first convolution layer presents redundancy ratio percentages up to 49%, 18%, and 26% for the experimented CNN models. Hence, the percentage of redundancy reveals high *intra-frame locality*. These charts also indicate that the same pattern of inputs and weights repeatedly appears for 49%, 18%, and 26% of the observed data in the first convolution layer, thus producing the same output patterns. Meanwhile, as the number of samples in the batch increases, the *inter-frame locality* comes up, and the redundancy rates grow too. When analyzing a batch composed of 10,000 images, the measurements of redundancy level for the first convolution layer reach percentages up to 76%, 92%, and 99% for the three different scenarios, respectively. Another important observation from these charts is that: the more in-depth the convolution layer is, the lower the measured redundancy tends to become. Also, deeper layers come with non-linearity and discretization processes inferred by the activation functions after the convolution process, such as upsampling, downsampling, and specific feature-extraction of subsequent filters.

As input data locality reflects into computation redundancy during the CNN execution, techniques such as computation reuse could exploit the intra-frame locality present in CNNs by keeping in near-storage previous convolution output results, thus leveraging computation reuse opportunities, and avoiding the costly dot-product operations. In the same way, techniques such as those based on video compression algorithms can be employed to exploit the inter-frame locality, bypassing the previous detection of a CNN when the similarity level between two frames is considered high.

3 RELATED WORK

In this chapter, state-of-the-art works regarding software and hardware endeavors to accelerate NNs are presented.

3.1 Custom hardware accelerators for NN execution

Due to CNNs increasing importance as state-of-the-art solutions for a large number of applications, several specialized hardware accelerators have been proposed in the past years. They aim to both accelerate and reduce CNN energy consumption by exploring intrinsic network characteristics, data redundancy, and similarity, as well as reuse opportunities. DianNao (CHEN et al., 2014a) is a machine learning accelerator focusing on high throughput and memory behavior, aiming to minimize and control memory transfers for large-scale networks. Still focusing on memory and large networks, DaDianNao (CHEN et al., 2014b) proposes a new custom-chip architecture. The Eyeriss (CHEN et al., 2017) accelerator is based on a row stationary data-flow to map CNNs computations into Processing Elements (PEs), minimizing data movements.

The accelerator proposed in Riera, Arnau and González (2018) applies linear quantization on the inputs to increase redundancy and combines this with a computation reuse technique that takes into account input similarity in each layer. The Unique Weight CNN Accelerator (HEGDE et al., 2018) exploits weight redundancy instead, reusing dot product results to reach a reduced network size, eliminating multiplications and limiting memory accesses. The technique proposed by Raha and Raghunathan (2017) is a quantized table lookup that approximates kernel functions. They reduce the number of values that need to be stored in the table by quantizing them to a smaller range, replacing expensive operations by table lookups. However, the operations they replace are on a smaller granularity level than our technique, referring to multiplications instead of whole convolutions.

LookNN (RAZLIGHI et al., 2017) replaces multiplication operations with lookup searches, relying on associative memory blocks to reduce power and execution time of Floating-Point Units (FPUs). LookNN's authors also provide an analysis of the additive error that their technique can introduce, and how to minimize it. The reconfigurable Bloom filter unit (JIAO et al., 2018) follows a similar approach, storing inputs and performing an approximate pattern matching strategy, avoiding execution on energy-

intensive FPUs. Inspired by these two previous work, the authors of Mocerino, Tenace and Calimera (2019) present a clustering method that maximizes the reuse property of a neural network and a CAM-enhanced floating-point that leverages operand-level reuse. Recently, Euphrates (ZHU et al., 2018) has been proposed, with an approach where the block matching technique replaces some CNN inferences, and predictions are made based only on estimated vector motion, which allows the expensive convolutions to be skipped.

3.2 Software-based techniques for efficient NN execution

Software techniques for CNN acceleration have also been recently presented, covering several different approaches. Quantized Neural Networks (HUBARA et al., 2017) reduce memory accesses and replace expensive arithmetic operations with bit-wise ones by training the networks with 1-bit weights and activations. The deep compression pipeline (HAN; MAO; DALLY, 2015) combines pruning, trained quantization, and encoding to obtain smaller networks while keeping the same accuracy levels. The Quantized CNN framework (WU et al., 2016) focuses on quantizing networks specifically for mobile devices. Winograd's minimal filtering algorithm and pruning are proposed in Liu et al. (2018), where they manage to increase weight sparsity to explore it further. These techniques work on available CNN models, modifying their structure to take advantage of network compression opportunities. These works reduce image inference time but are not ideal for videos as they do not consider image similarity.

MobileNets (HOWARD et al., 2017), on the other hand, comprises specialized network architectures that focus on mobile and embedded vision applications. They are designed to be small and present low latency, according to the limitations of target applications. The Fast YOLO framework (SHAFIEE et al., 2017) combines an optimized network architecture for mobile devices based on YOLOv2 (REDMON; FARHADI, 2017) and an inference module. They aim to perform real-time object detection in videos with feasibility for mobile devices. Those two approaches require training new models, and so cannot be directly applied to existing working networks.

To optimize video inference, NoScope (KANG et al., 2017) trains shallow Neural Networks (NNs) as specialized models of a target network. Those models are smaller and can only identify a limited set of objects they have been trained to, out of the original objects the reference network is trained for. Thus, shallow NNs are less computationally expensive and ideal for applications that do not need full network capabilities. A differ-

ence detector mechanism is also used: if frames are deemed too similar, no new prediction will be computed. However, this method does not generalize to videos other than the ones they were targeted to classify and hence lacks generality.

The work 3D CNNs (JI et al., 2013) approaches the video analysis problem by performing convolutions on multiple stacked frames to explore the temporal locality properties of inputs. Those networks, however, are focused on accuracy performance, instead of energy reduction and network compression. Video classification with CNNs (KARPATHY et al., 2014; NG et al., 2015) aims to categorize full videos based on what is happening in them, instead of individually classifying objects within each frame.

3.3 Work comparison

To summarize the main aspects found in the state-of-the-art works that accelerate CNN execution, we organized some features from the related work in Table 3.1. The cells marked as "✓" indicate that the feature is provided.

Regarding the related accelerators (on the top of the table), although they have reached significant improvements in execution time and energy consumption reduction, they do not cover all the granularities (multiplications, convolutions, and inferences) reductions as this work does. Despite Mocerino, Tenace and Calimera (2019) have achieved energy savings up to 70% (21.6% of improvement compared to Jiao et al. (2018)) for the MNIST dataset running in a pre-trained model of the LeNet CNN, their results make mention only to the energy spent on FP units. In contrast, the results achieved by this work for the same experiment, report energy savings regarding the whole computer system up to 48% for the convolution kernel-level reuse and 74% for the grid-level reuse. Moreover, these works introduce several drawbacks for software and hardware design. For the software design field, they cause new challenges to the programmability task. Legacy programs must be redesigned to cover the new hardware's intrinsics such as memory mapping, communication protocols, etc. For the hardware project and manufacturing processes, there is a noticeable increase in production cost. Nonetheless, our work differs by being a software-only solution able to run on GPPs while achieving competitive gains. Furthermore, we do not focus only on one technique, but rather on applying both computation reuse and motion prediction wherever and whenever they are most needed.

Compared to state-of-the-art algorithm solutions for CNN execution, our approach differs from other works, where the most common approach employed is the quantization

or transformation of the original data to reduce the amount of data transferred and the number of multiplications. Additionally, despite those works that propose specific domain smaller NNs, reducing the number convolutions required without big accuracy drops, they need the new CNN model to be retrained.

Table 3.1: Related work comparison

	ASIC	Less memory transfers	Less #Mult	Less #Conv	Less #Inferences	Network retraining
Chen et al. (2014a)	✓	✓	✓	✗	✗	✗
Chen et al. (2017)	✓	✓	✓	✗	✗	✗
Riera et al. (2018)	✓	✓	✓	✗	✗	✗
Hegde et al. (2018)	✓	✓	✓	✗	✗	✗
Raha et al. (2017)	✓	✓	✓	✗	✗	✗
Razlighi et al. (2017)	✓	✓	✓	✗	✗	✗
Jiao et al. (2018)	✓	✗	✓	✗	✗	✗
Mocerino et al. (2019)	✓	✗	✓	✗	✗	✗
Zhu et al. (2018)	✓	✓	✗	✗	✓	✗
Hubara et al. (2017)	✗	✓	✓	✗	✗	✗
Han et al. (2015)	✗	✓	✓	✗	✗	✗
Wu et al. (2016)	✗	✓	✗	✗	✗	✗
Liu et al. (2018)	✗	✓	✓	✗	✗	✗
Howard et al. (2017)	✗	✓	✓	✓	✗	✓
Shafiee et al. (2017)	✗	✓	✓	✓	✗	✓
Kang et al. (2017)	✗	✓	✓	✓	✗	✓
Ji et al. (2013)	✗	✓	✗	✗	✓	✗

Source: Author

4 PROPOSED APPROACH

In this section, we exploit the computation reuse opportunities previously pointed out in Section 2, which come up with the CNNs inference due to existing intra-frame and inter-frame localities in different datasets. We address the intra-frame locality with the employment of a computation reuse technique to replace convolution operations by memory accesses and retrieve previously-stored results. Although computation reuse is an alternative to avoid costly dot-product operations with repeated patterns, the benefits on energy savings may not be that intuitive. Replacing a floating-point multiplication by a table lookup can improve energy efficiency to some extent, but there is a narrow design space to trade accuracy by energy-saving using operand-level reuse (RAZLIGHI et al., 2017; JIAO et al., 2018; MOCERINO; TENACE; CALIMERA, 2019).

The main drawback of multiplication reuse remains on the increased energy consumption due to on-chip memory accesses. For instance, a 16-bit fixed-point multiplication in 32 nm consumes 0.4 pJ, whereas the corresponding lookup table access costs 2.5 pJ, regarding a 32K-entry 16-bit Static Random Access Memory (SRAM) (HEGDE et al., 2018; MURALIMANOHAR; BALASUBRAMONIAN; JOUPPI, 2009). However, by extending the idea of computation reuse backed on the results presented in Figure 2.3, we can reduce the number of multiple lookups by a single search for each convolution, thus opening up opportunities to reach more significant energy savings. This advance on the degree of reuse, from multiplication-level to convolution kernel-level granularity, implies in changes to the CNN algorithm. By replacing multiplications and additions, we may reduce the costs of memory access in comparison to previous work and also make it feasible for CPUs and GPUs without any hardware modification.

To store and, further, retrieve convolution results from memory, we propose a hash table scheme to associate any arbitrary input with its corresponding output value for each convolution layer's filter. Further, we demonstrate by measuring the amount of memory required to keep all the output values in storage to be disadvantageous, even when considering duplicated values due to redundancy. To overcome the high lookup table sizes, we perform statistical analysis over the stored data, and we develop a magnitude-based clustering mechanism to reduce the table sizes, making the employment of computation reuse for the CNN realm feasible. Finally, we expand the reuse granularity from kernel-level to the convolution grid-level, where groups of adjacent activations in the feature map can be stored for further reuse.

Regarding the inter-frame locality, we exploit the frame-level of reuse backed on a block matching-based technique to exploit frame similarity between inputs in camera videos. The block matching is widely used for video compression algorithms, where it predicts motion between two sequential frames and stores the prediction as a Motion Vector (MV), further to reduce the space storage. In this work, we modify the original CNN inference to bypass the last output inference if the current input frame is similar to the previous one, thus skipping the CNN call operation.

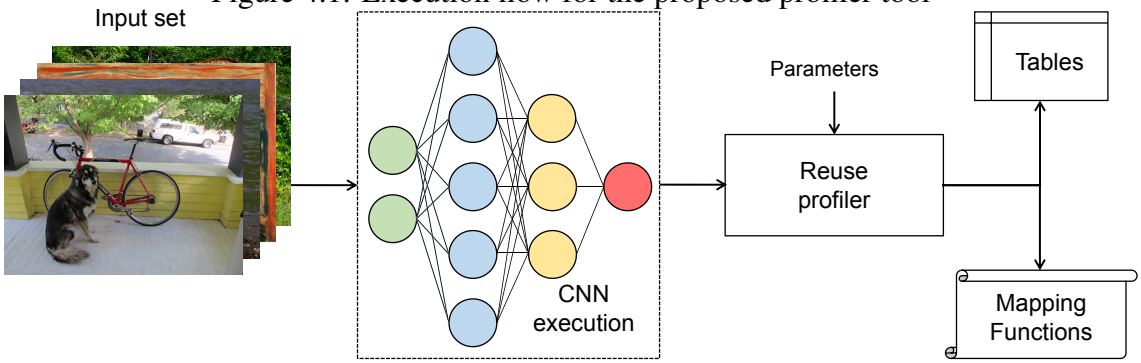
4.1 Convolution kernel-level reuse

To employ a reuse technique to keep in near-storage all the convolution output values given a CNN execution, we associate every pair of input and feature map weight operands related to a convolution, which outputs a single outcome value ($(I_i, W_k) \rightarrow O_{ik}$) with a hash table access. Since a convolution can be represented by a multivariate polynomial function ($F(I_i, W_k) \rightarrow O_{ik}$), and to maintain the injective relation between the domain (I_i, W_k) and the codomain (O_{ik}), we split the output set into M subsets. Each new subset comprises a hash table that contains all the convolution values generated by the combination of the input (I_i) with the respective kernel weight (W_k). By splitting the output set into smaller ones, it is possible to replace the multivariate polynomial function by only one-variable dependant (only the input I_i), and potentially less computational costly, mapping function recreating the relation that matches each I_i element to its corresponding output value given a constant weight kernel value ($F_k(I_i) \rightarrow O_{ik}$). Since we employ filter-sized granularity of reuse over any input feature, a filter-sized input has to be reduced to a single key using a Cyclic Redundancy Check (CRC)-inspired algorithm. Then, the index is used to retrieve the output value associated with that input in a hash table of a given filter. In other words, the algorithm now skips convolution over an input volume performing CRC and hashing to find a value within the hash tables for each different filter.

Further, implementing such a reuse technique requires a profiling tool that generates the hash tables before the inference process. First, the tool must run a batch of inputs and collect all the tuples $< I_i, W_k, O_{ik} >$ that match each convolution operation. Then, the *profiler* processes the data, generating the *Output tables* and the *Mapping functions* files. The *Output tables* are hash tables that contain all the output values (O_{ik}) occurrences for each kernel (k) observed during the profiling time. The *Mapping Functions*

are indexing functions in charge of coordinating the appropriate mapping for any input (I_i) to its corresponding output value (O_{ik}) in the corresponding hash table. Initially, to retrieve an output value for each filter's table, we employ a CRC-like mapping function over the input (I_i)to reduce the input size, and also to generate a hash index to consult the table. We have chosen a CRC-like function since it does not perform any transformation in the data, and it comprises only bitwise operations, which are meant to be less computationally expensive than the original convolutions. Further, other more adequate mapping functions could be employed, possibly to better exploit some input property or behavior. Figure 4.1 illustrates the execution flow of the proposed profiler tool.

Figure 4.1: Execution flow for the proposed profiler tool

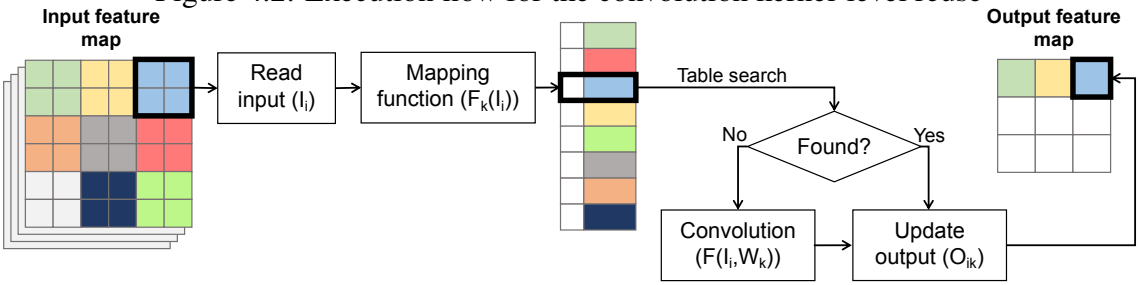


Source: Author

Now that we have the *Output tables* and the *Mapping Functions* generated by the profiling tool, the execution flow for each filter of each convolution layer follows the steps described in Figure 4.2. First, the input feature map (I_i) is read, then the corresponding mapping function (F_k), which matches the current filter (W_k) in the *Mapping Functions* set generates a key to search in the hash table. After getting the hash key, we try to retrieve the entry in the hash table, which matches the current input feature map to its respective output value (O_{ik}). If we succeed in searching the table, the corresponding output feature map is updated with the value brought from the table. Otherwise, the current output value must be calculated by a convolution. Additionally, when the lookup table search retrieves a mismatch, we opt not to update the table, since it may incur modifications in the *Mapping functions* that are statically created during the *Computation reuse profiler* execution. Moreover, we conducted several experiments and found that even a static table, keeping only exact matches, can reach hit ratios up to 97%, when performing inferences over different images than the ones used during the profiler phase.

However, it is only viable to keep in memory a smaller set of a large dataset like MNIST, ImageNet, and CAVIAR. Hence, the profiling dataset for computation reuse must provide a representative sample of the whole datasets. Despite an input set of 10,000 ran-

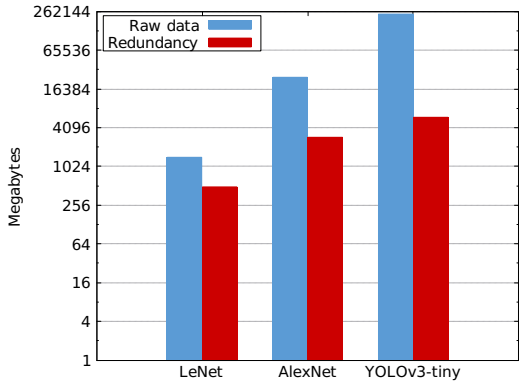
Figure 4.2: Execution flow for the convolution kernel-level reuse



Source: Author

dom images from the datasets mentioned above, the profiling tool generates large hash tables, which require up to 1.4 GB, 24.7 GB, and 235.1 GB of storage for running the three datasets in the LeNet, AlexNet, and YOLOv3-tiny CNNs, respectively. Storing all the convolution output values for a given batch can be a limiting factor for this type of implementation since it is a one-to-one relationship of (I_i, W_k) and O_{ik}, which may introduce repeated entries in the lookup table. Further, even considering the repetition of inputs that causes redundancy in the CNN execution, the amount of memory to store all the non-repeated output values for each layer's filter might be still impracticable. Figure 4.3 illustrates the table sizes in megabytes required to store all the memorized values for the execution of 10,000 images from MNIST, ImageNet, and CAVIAR datasets running on LeNet, AlexNet, and YOLOv3-tiny CNNs, respectively. The label *Raw data* refers to the table sizes without considering repeated values in the input set when the label *Redundancy* does. It is possible to notice a reduction in the table sizes from 1.4 GB to 489 MB, 24.7 GB to 2.8 GB, and 235.1 GB to 5.9 GB for the three experimented scenarios.

Figure 4.3: Table size (Megabytes) to keep in storage convolution values for a batch of 10,000 inputs



Source: Author

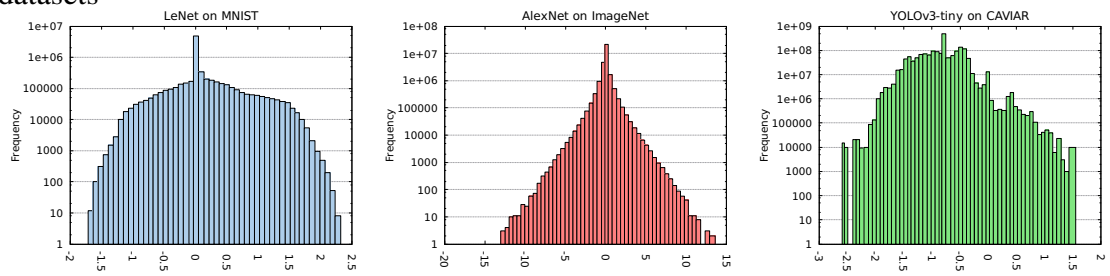
4.1.1 Performing data analysis to reduce table size

Managing tables containing all the output values profiled to exploit computation reuse throughout the CNN execution requires a considerable amount of memory, as demonstrated in figure 4.3. Even the 489 MB of space related to the execution of the LeNet CNN over the MNIST dataset can be prejudicial for the proposed lookup technique. Performing a lookup for a result in a 489 MB table size may introduce drawbacks in both performance and energy gains since the searched data is likely not to be found in any cache level in the whole memory hierarchy, though in the Dynamic Random Access Memory (DRAM) module. Moreover, the time and energy costs associated with DRAM access can reach thousands of magnitude when compared to first-level cache constraints. For example, a 4 GB DDR4 memory consumes approximately 16.7 nJ of energy per reading operation, and it takes roughly 30 ns to deliver data to the Last-Level Cache (LLC). Hence, it is mandatory to keep all the tables as near as possible to the CPU to avoid the costly data transfer between the memory levels.

One plausible solution to keep the tables closer to the CPU is by reducing the table sizes. State-of-the-art works that employ memoization or lookup tables to hold previous results have reduced their memory footprints by either cutting bits in the input (JIAO et al., 2018) or by delivering quantization techniques in the activation kernel weights (MOCERINO; TENACE; CALIMERA, 2019), allowing clusterization of the data. Likewise, but in a different scope, this work conducts statistical analysis over the convolution outputs gathered during the profiling phase. Figure 4.4 presents the output values histogram for the first filter of the first convolution layer for several CNN models running different datasets on a batch size of 10,000 inputs. One can notice that all the charts resemble a normal distribution. Exploiting this central tendency behavior in the data in favor of the reduction of the table sizes can be advantageous for the proposed technique for two main reasons:

1. The *central limit theorem* ensures that even when unknown random variables are added to a set, they tend toward a normal distribution. Hence, a new input pattern will likely match one of the already mapped inputs
2. The values present magnitude proximity, which allows us to perform clustering and approximations of the observed data further to reduce the table sizes without hurting the data's representativeness.

Figure 4.4: Histogram of convolution outputs for three CNN models with different datasets



Source: Author

Since the output convolution values tend to a normal distribution as depicted in Figure 4.4, a clustering algorithm could be applied in the data to reduce the number of activations to store. Heuristics such as *Jenks Natural Breaks*, *k-means*, and *mean-shift clustering* have been used to perform clustering in CNN domains (MOCERINO; TENACE; CALIMERA, 2019; GONG et al., 2014; HAN; MAO; DALLY, 2015). However, these approaches estimate their cluster centroids based on metrics, such as the frequency distribution or the variance of the values, that overlook a crucial data conditioner that commonly comes up after the convolution process: the *activation function*. Activation functions usually take place to insert non-linearity in the data after the convolution layers by cutting negative values or smoothing them. Advanced CNN models rely on rectifiers like *ReLu*, *Leaky ReLu*, and even *Linear* functions, since they are much faster in terms of execution time than saturating nonlinearities, such as sigmoid and hyperbolic (REDMON; FARHADI, 2018; REDMON; FARHADI, 2017; SZEGEDY et al., 2017). Thus, a clustering mechanism that exploits the activation function behavior could achieve better results in terms of table size reduction and better data representativeness.

Thereby, this work introduces a clustering mechanism based on the activation function that groups entries by its proximity and provides smaller lookup tables. Initially, the mechanism covers only the *ReLu*, *Leaky ReLu*, and *Linear* variations since they are the most common types found in recent CNN models. The practical implementation of the clustering requires modifications to the profiling tool presented in Figure 4.1. The main parameters of the clustering mechanism are:

- **O**: The values gathered from a convolution layer's output feature map execution.
- **RangeDistance**: The range distance that determines the approximation percentage between the clusters.
- **RectifierType**: The activation function type that comes up after the current convolution layer.

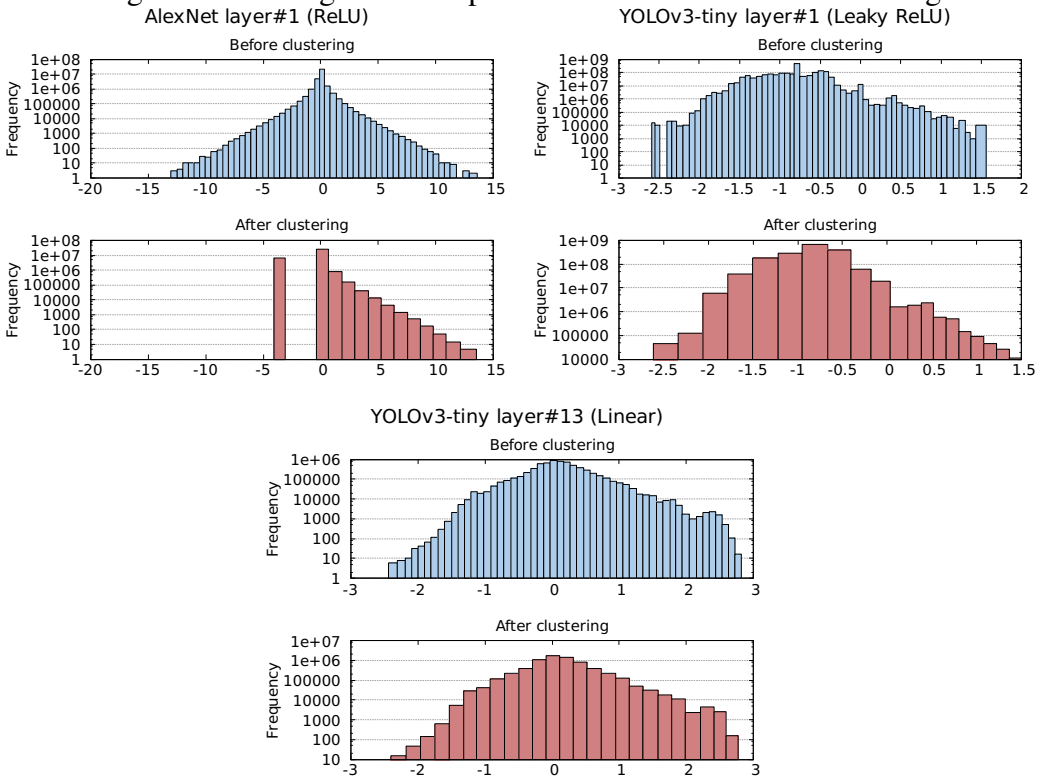
- **MultiplierFactor**: The multiplier factor is applied over the RangeDistance depending on the RectifierType.

The whole clustering mechanism is completed in a four-step procedure:

1. First, all the output values (O) are sorted in ascending order.
2. Then, clusters are created, where each number is enclosed within an interval defined by a temporary variable called ClusterDistance. The ClusterDistance is determined accordingly to the RangeDistance and the current activation function type: If the function is a Linear activation, the ClusterDistance is set to the RangeDistance value for all the numbers. However, if the function is a ReLu rectifier, all the negative values are automatically mapped to their average value, and the ClusterDistance is set as the RangeDistance (Rd) for all positive numbers. Otherwise, if the function is a Leaky ReLu rectifier, the ClusterDistance for the negative values is set to the RangeDistance multiplied by the MultiplierFactor, and it is kept as the RangeDistance for the positive ones.
3. Thirdly, the mechanism replaces all the convolution values inside each cluster by their average value.
4. Finally, as a result of the clustering, the former one-to-one relationship of (I_i, W_k) and O_{ik} is replaced by a many-to-one (I_i, W_k) and $\bar{O}_{ik}$, where the $\bar{O}_{ik}$ as the output indicates that several inputs new are mapped to the same output convolution value. Hence, both the *Tables* and the *Mapping Functions* in Figure 4.1 are updated to generate the correct output values.

Figure 4.5 presents the output tables histogram comparison before and after the execution of the proposed clustering mechanism. There are presented examples of convolution layers extracted from several CNNs, followed by three different types of rectifier functions. For all the charts showing the reduced tables (red bars), the RangeDistance value was set to 4%, while the MultiplierFactor was set as $2\times$. For the first convolution layer in the AlexNet CNN, which is followed by a ReLU rectifier function, it is possible to notice that all the negative values were mapped to a single red bar, while the positive ones were mapped to clusters with the same Cluster Distance. The conditioner function type related to the first YOLOv3-tiny's convolution layer is a Leaky ReLU variant. Thus, all the negative values are enclosed in broader ranges than those positive. Finally, YOLOv3-tiny's convolution layer #13 comprises a Linear rectifier. Hence, the Clustering Distance is applied equally over all the range of numbers.

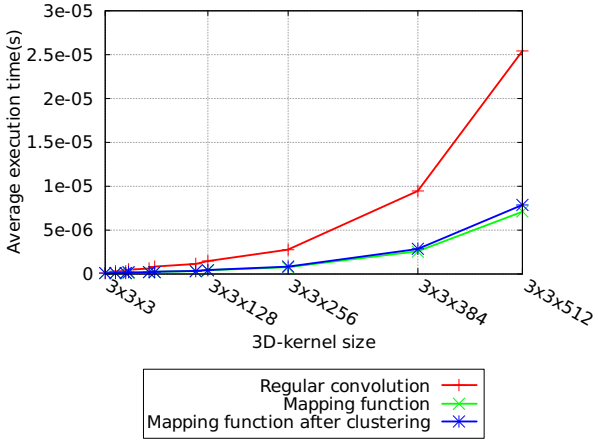
Figure 4.5: Histogram of output values before and after clustering



Source: Author

To correctly coordinate the new *Mapping Functions* execution, we add to the original CRC implementation a bitmask operation, which works as a truth-table, selecting only the input bits whose variation changes the output values, thus ignoring the don't care bits. This approach is similar to those presented by Jiao et al. (2018) and Raha and Raghunathan (2017), whereas we approximate the values based on the output result further to select the bits to ignore in the input. To illustrate the costs related to perform an index calculation versus a regular convolution, Figure 4.6 presents the average execution time spent to calculate a convolution and to perform index calculations over different 3D input sizes. On average, an index calculation is $4\times$ faster than a regular convolution, even considering a small overhead due to the bitmask after the table clustering (Label Mapping function after clustering). Hence, if we can reduce the table sizes and consequently their access time, significant energy savings, and inference time reduction can be achieved, as will be presented in Chapter 5.

Figure 4.6: Mapping Functions and regular convolution execution time comparison



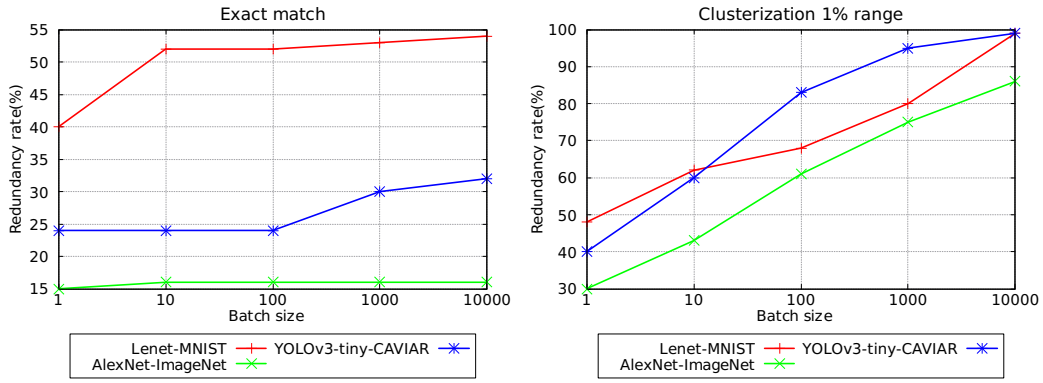
Source: Author

4.2 Grid-level reuse

Real-world images present interesting characteristics related to the distribution of pixels still untapped for computation reuse. We are interested in two properties of these images: a) the recurrent pixels to form an object, and b) the probability distribution of image pixels. Backed on these properties, we may want to predict the next activation based on past, nearby values. Firstly, recurrent pixels are undoubtedly present even after image segmentation. Most of the objects from accessible datasets have regions with repeated pixels and shades in their surroundings, thus generating the same pattern of activations. In this situation, the kernel-level technique already benefits enormously from the temporal locality. Also, the second property enables us to interpret images as samples from a high-dimensional probability distribution, which is broadly explored in generative models (GOODFELLOW et al., 2014). Thus, such information can be used to improve entries' allocation in the memory or also load probable activations ahead of computing their hash indexes.

Figure 4.7 demonstrates the percentage of repeated sequences composed of four neighboring activations (2x2 submatrices) in feature maps of different CNN models. Though the reuse rate may depend on the CNN and dataset, from 15% to 40% of the activations, on average, repeatedly appear in the feature maps of a single image. As we increment the number of inferences, the redundancy rate can reach up to 54%, as shown in the leftmost chart of Figure 4.7. Moreover, the potential for reuse can reach 99% when we consider approximate values obtained by clustering within ranges of 1%, as presented in the rightmost chart of Figure 4.7. Here, the test case scenario plays a vital role in grid reuse, since the highest reuse rate occurs on CAVIAR and MNIST datasets. These sce-

Figure 4.7: Percentage of repeated 2x2 submatrices throughout the layers of different CNN models and datasets



Source: Author

narios are composed of video recording with a static background and hand-written digits with black and white pixels, respectively.

Thus, the tool has to find the arrangement of adjacent activation most likely to happen. This process is easily automated by changing the profiling flow presented in the preceding section and by providing parameters such as the structure of recurrence (matrix or vector) and thresholds for table filling. Also, the table structure has to be changed and a trade-off between table size and number of adjacent values to be stored. Here, we are going to consider that each entry in the table is composed of four activations. This initial setup is enough for demonstrating the grid-level of reuse potential while keeping it feasible with low overhead in table size. The hashing function still maps to the approximate value obtained in the clustering process, which is the first value of the entry. However, it is now followed by the most probable left and right bottom, and top-right activations. Then, we may take advantage of this information for speculatively loading the next activations.

Given a filled lookup table, one may use adjacent values in the hash table to exploit the spatial locality without modifying the algorithm presented in the last section. However, mapping those neighboring activations to contiguous addresses is a challenging task for any tool. Another way to use the sequence of activations is by loading the next activation ahead of computing the index associated with that activation. In this approach, the activations loaded ahead of its time can be seen as an approximate technique, since we use them to skip the hashing operations of the algorithm flow presented in Figure 4.2. An improvement for this algorithm is shown in Procedure 1. This routine enables us to skip some operations if we accept some uncertainty in partial results, but also enables us to verify them later with overlapping activations. Also, Figure 4.8 illustrates some iterations of the execution flow.

Procedure 1 *Compute feature maps using kerne-level and grid-level reuse*

Input: $InputFMap, Table, InputHeight, InputWidth, KernelSize$
Output: $OutputFMap$

```
1:  for y ← 0 to InputHeight do
2:    for x ← 0 to InputWidth do
3:      actvs ← Hash(Table, CRC(InputFMap(y, x)))
4:      for kh ← 0 to KernelSize do
5:        for kw ← 0 to KernelSize do
6:          OldActv ← OutputFMap(y + kh, x + kw)
7:          if OldActv ≠ actvs[kh * KernelSize + kw] or isNonValid(OldActv)
            then
8:            OutputFMap(y + kh, x + kw) ← actvs[kh * KernelSize + kw]
9:          else
10:           actvs2 ← Hash(Table, CRC(InputFMap(y + kh, x + kw)))
11:           for kkh ← 0 to KernelSize do
12:             for kkw ← 0 to KernelSize do
13:               OutputFMap(y + kh + kkh, x + kw + kkw) ←
                    actvs2[kkh * KernelSize + kkw]
14:             end for
15:           end for
16:         end if
17:       end for
18:     end for
19:   end for
20: end for
21: return OutputFMap
```

Initially, we apply the CRC and hash functions to obtain the entry related to a submatrix of an input image or a feature map. Next, we retrieve the activation value along with its most recurrent neighbors (line 3 in Procedure 1). This compound entry contains four activation values: the top left, which generates the hash index, followed by the top right and bottom values, which are stored speculatively in their respective positions. The step ❶ in Figure 4.8 illustrates the mapping of convolution from the input (gray) to the output (medium blue), followed by the supposed activations (light blue).

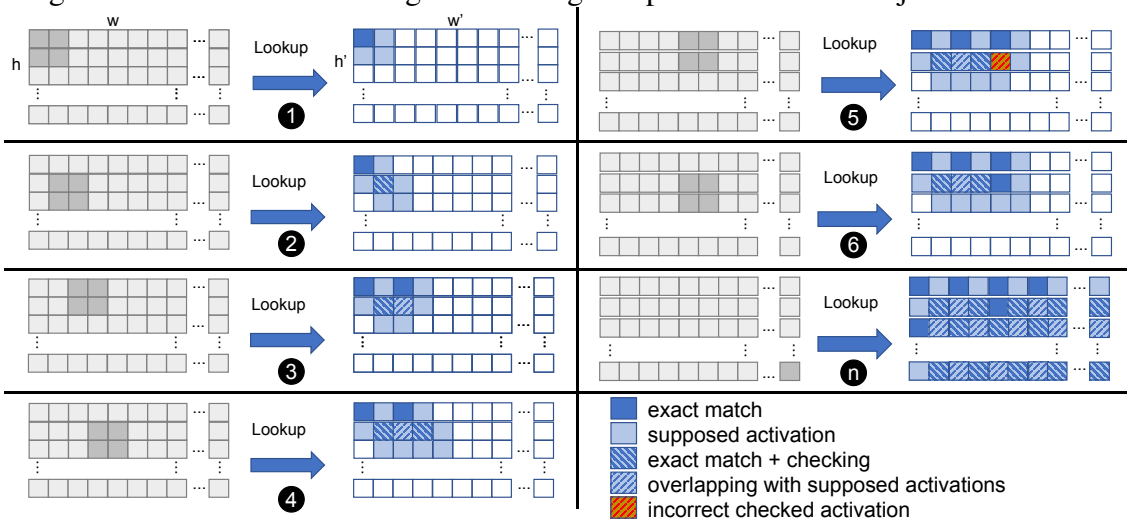
Then, we proceed to the next iteration (step ❷) where the hashing function is applied to another submatrix of the input that overlaps the previous region. The lookup search is performed not only to retrieve a more precise activation for the current position of the output feature map but also to verify if part of the last prediction was accurate enough (line 7 in Procedure 1). If the overlapping values (blue patterned) match, we proceed to the next 2x2 submatrix (lines 10,11,12 and 13). It is important to notice that the main loop computes over three input rows at each time, which maintains the streaming behavior of the input feature.

Then, the next iteration (step ❸) is similar to the step ❶, except for an additional operation that checks two supposed activations in the overlapping region (light blue patterned). From step ❶ to ❹, we capture the behavior of highly predictive speculation. When a mismatching of overlapped values happens, as illustrated in step ❺, a new lookup for that position is made. In the last step (ⓝ), one can notice that the upper row and left column have values which are not checked against other speculative loads, thus increasing error to our execution. However, the majority of the supposed activations are checked against more precise, nearby activations. Therefore, this algorithm enables us to reduce up to 50% the number of hash operations in the best-case scenario for a 2x2 grid-size. Although the mismatching does not represent a significant penalty in execution time, not always our prediction is precise and the savings can reach 50% when compared to the previous algorithm using only kernel-level reuse. Thus, we present the energy and time savings in the next section, as well as accuracy and average table size required for these techniques.

4.3 Frame-level reuse

The inter-frame locality in videos has already been exploited in the past by video compression techniques. The notion that frames change gradually over time, instead of

Figure 4.8: Iterations of the algorithm using computation reuse of adjacent activations



Source: Author

abruptly, is natural for videos. Based on that, video encoding standards commonly perform motion estimation between frames to further improve compression rates. To perform motion prediction between frames, the block matching algorithm is widely employed (JAKUBOWSKI; PASTUSZAK, 2013).

The block matching technique works by dividing a frame in macroblocks of size MxM, and searching in the previous frame for the corresponding block that minimizes matching error, over a search area of size N. The displacement between two same macroblock regions present in different frames is referred to as the Motion Vector. An MV for each macroblock indicates where the macroblock came from in relation to the previous, comparison frame; in this way, motion can be predicted over time. Block matching algorithms try to find the best matching block from the previous frame by minimizing matching error. The matching error between the block at position *(x,y)* in the current image, I_t, and the candidate block at position $(x+u, y+v)$ in the reference image, I_{t-1}, is usually defined as the Sum of Absolute Difference (SAD), where the block size is MxM (CHEN; HUNG; FUH, 2001).

$$SAD_{(xy)}(u,v) \equiv \sum_{j=0}^{M-1} \sum_{i=0}^{M-1} |I_t(x+i, y+j) - I_{t-1}(x+u+i, y+v+j)| \tag{4.1}$$

The offset between where a block was in the previous frame and where a block is in the current one determines its MV. The best estimation of the MV, $(\hat{u}, \hat{v})$, is set as (u, v), which minimizes $SAD_{(x,y)}(u,v)$ The specific block matching algorithm to estimate the

$(\hat{u}, \hat{v})$ used in this work is Three Step Search (TSS), which was modified from the code provided by Chen, Hung and Fuh (2001). Nevertheless, since its execution is independent of the network, any variant could be used in its place.

Figure 4.9: CNN and motion prediction comparison.

a) Regular CNN prediction of frame *f* b) Regular CNN prediction of frame *f+1* c) Motion prediction of frame *f+1*

The combination of motion estimation and regular video inference comes quite naturally. Previously CNN-predicted bounding boxes can have their new locations updated according to calculated frame motion. By checking the macroblocks that overlap predicted bounding boxes, one can find the associated MV of each bounding box. This process eliminates the need for CNN inference during several frames. Figure 4.9 shows a comparison between running the CNN and performing motion prediction. The first frame is predicted by the network, which is the same for both techniques. For the second frame, however, motion prediction does not need the CNN inference and achieves similar predictions.

Since applying motion prediction to several consecutive frames can accumulate errors, we take the SAD metric into account to decide whether the network will need to run the CNN or not. When the error is less than a predefined threshold, the motion prediction is used to update the output prediction frame. Otherwise (when the error is greater than the threshold), the motion prediction data is discarded, and a regular CNN inference is done instead. To control the updated bounding box sizes, we have also set minimum and maximum size values that they can have: when the predicted bounding box size is not enclosed between the minimum and maximum values, a CNN inference must be called for the frame too.

The proposed modifications for the MV algorithm follow the steps described below:

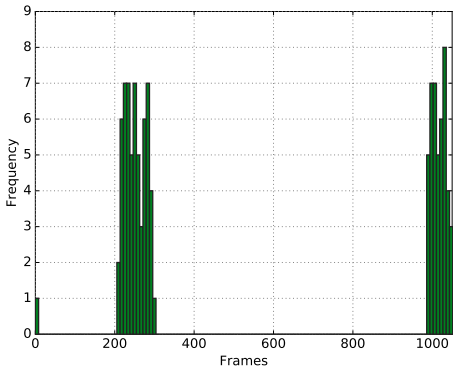
1. The CNN is called for the first video frame.
2. The next frame *f* is fetched and the block matching technique is applied considering *f* and *f-1*.

3. If the SAD metric is greater than a set threshold, or the bounding boxes are of invalid sizes, the CNN is called for the frame *f*. If not, then the motion prediction technique is applied to the frame *f*.
4. Go back to step 2.

Regarding step 3, the threshold value must be calculated separately depending on the camera. For example, the threshold necessary for an autonomous car application is different than one needed for indoor cameras. This happens since luminosity, camera focus, FPS, among other variables, change from camera to camera. We discuss how this work uses thresholds, and also how they have been computed in Section 5.

Figure 4.10 shows that there is a huge potential for energy savings in video inference. The figure's histogram illustrates the behavior of the proposed technique when running a set of 1.050 frames, which comprises 42 seconds of a video, in the YOLOv3-tiny network. The video chosen is the same as presented in Figure 4.9. As shown, the first frame is always executed. After that, the chart has CNN execution spikes that correspond to sections of the video where objects are moving. These spike regions, however, still do not have the same computational demand as a full CNN run would have, as seen in the irregular pattern present. Further, even when motion is present, we can apply block matching to several frames.

Figure 4.10: CNN execution histogram distribution



4.3.1 Execution flow

By combining the two approaches proposed in Sections 4.1 and 4.3, our method cannot only drastically reduce the number of times that the image inference needs to be performed, but also reduce each inference's cost. Figure 4.11 illustrates the execution

flow when combining the two former techniques. Each step works as follows:

Load first frame. The very first frame of the video is loaded. This frame always needs to be analyzed by the network in order to have the first reference prediction.

CNN computation reuse. Performs the CNN inference on a frame. Following 4.1, we replace a regular network with our computation reuse technique.

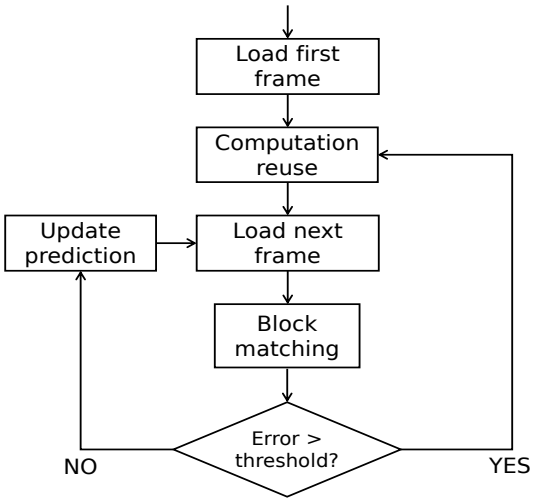
Load next frame. Fetches the next frame to be analyzed.

Block matching. Applies our block matching strategy to current frame *f* and previous frame *f-1*. This step calculates new bounding boxes positions if any were present in frame *f-1*.

Error > threshold? Based on the SAD value obtained in the block matching step and on the new bounding boxes sizes, decides whether a full inference is needed on frame *f*. If so, then the new bounding boxes values are discarded, and we go back to the CNN computation reuse step. Otherwise, we go to the update prediction step.

Update prediction. The bounding boxes placements calculated in the block matching step are used as the inference result for the current frame *f*. This frame never goes through the network.

Figure 4.11: Complete execution flow.



5 EVALUATION AND ANALYSIS

To evaluate the proposed techniques presented in Chapter 4, we implemented the execution flows described in Figures 4.1, 4.2, 4.8, and 4.11 in the Darknet framework. The Darknet is an open-source neural network framework written in C and CUDA (REDMON, 2013–2016). The standard data representation used in Darknet is the IEEE 754 single-precision floating-point format. Additionally, to separately measure the impact of each one of the three granularities of reuse addressed in this work, we built different scenarios, one for each technique exploration. Along with the experiments, we selected the datasets MNIST, ImageNet, CAVIAR, Caltech's Pedestrian Detection benchmark (DOLLÁR et al., 2009), and the GTEA dataset (FATHI; REN; REHG, 2011; LI; YE; REHG, 2015). Regarding the NNs employed in this work, we ran the abovementioned datasets in the CNN models LeNet, AlexNet, YOLO3-tiny, YOLO3-tiny, and YOLO3-tiny, respectively.

The MNIST dataset is comprised of handwritten digits, where all the digits have been size-normalized and centered in a fixed-size image. This dataset is an excellent alternative for people who want to try machine learning algorithms without spending lots of computational efforts. The ImageNet is a dataset built to provide variety and dense coverage of the image world. It organizes its different classes of images in a densely populated semantic hierarchy composed of 12 subtrees and 5,247 categories, varying for mammals to vehicles. The CAVIAR dataset comprises video frames recorded over different scenarios. These scenarios include people walking alone, meeting with others, entering and exiting compounds, etc. The Caltech's Pedestrian Detection benchmark is a dataset that contains video frames collected from a moving vehicle. These videos include several traffic scenarios with frequently occluded people. We have chosen these last two datasets for our experiments since they comprise of real-life situations specific to two essential applications: security cameras and autonomous cars. Moreover, with the GTEA dataset, which comprises videos from a camera attached to a cap worn by a person, we cover situations with daily activities.

All baseline results refer to pre-trained models of the tested CNN architectures and datasets, running in the original version of the *Darknet* framework and in the system processor system described in Table 5.1. We based our execution time and energy consumption estimations on hardware counters (AVG clock frequency, clock cycles, and cache statistics) available in the processor described in Table 5.1, and power constraints found in the user's manual guides for both the CPU and the DRAM module. The com-

plete description regarding each evaluation scenario, such as the number of evaluated frames and parameters for the computation reuse tool, is done at the beginning of the corresponding section of each computation reuse granularity technique.

Table 5.1: System configuration.

CPU processor
Intel(R) Core(TM) i5-7500 CPU; 4 cores; 3.40GHz;
AVX2 Instruction Set Capable;
32KB IL1 cache; 32KB DL1 cache;
1MB L2 cache;
6MB L3 cache;
Total Power - 65W;
DRAM
DDR4 2133MHz;
Total DRAM Size 16GB;
Total Power - 6.4W;

Source: Author

5.1 Convolution kernel-level reuse

To evaluate the technique presented in Section 4.1, which address computation reuse at convolution kernel-size granularity, we employed the datasets MINIST, ImageNet, and CAVIAR, running in the CNNs LeNet, AlexNet, and YOLOv3-tiny, respectively. Moreover, we ramdonly selected 10,000 images for each dataset as input for the *Computation Reuse* tool. For evaluation, we selected 6,000 frames from each dataset. For MNIST and ImageNet, we randomly select 6,000 images from each dataset as validation scenarios. From CAVIAR, we selected videos of two different surveillance cameras, each recording a different corridor angle of a shopping center. To illustrate how the presented technique impacts the accuracy, the lookup table sizes, the inference time, and energy consumption, we have chosen three different (5%, 10%, and 20% for LeNet; 0.05%, 0.13%, and 0.2% for AlexNet; 0.05%, 0.1%, and 0.13% for YOLO) *RangeDistance* values for each dataset and CNN pair. As the *MultiplierFactor* for the clustering technique presented in Section 4.1.1, we set it as $2\times$ for all the experiments.

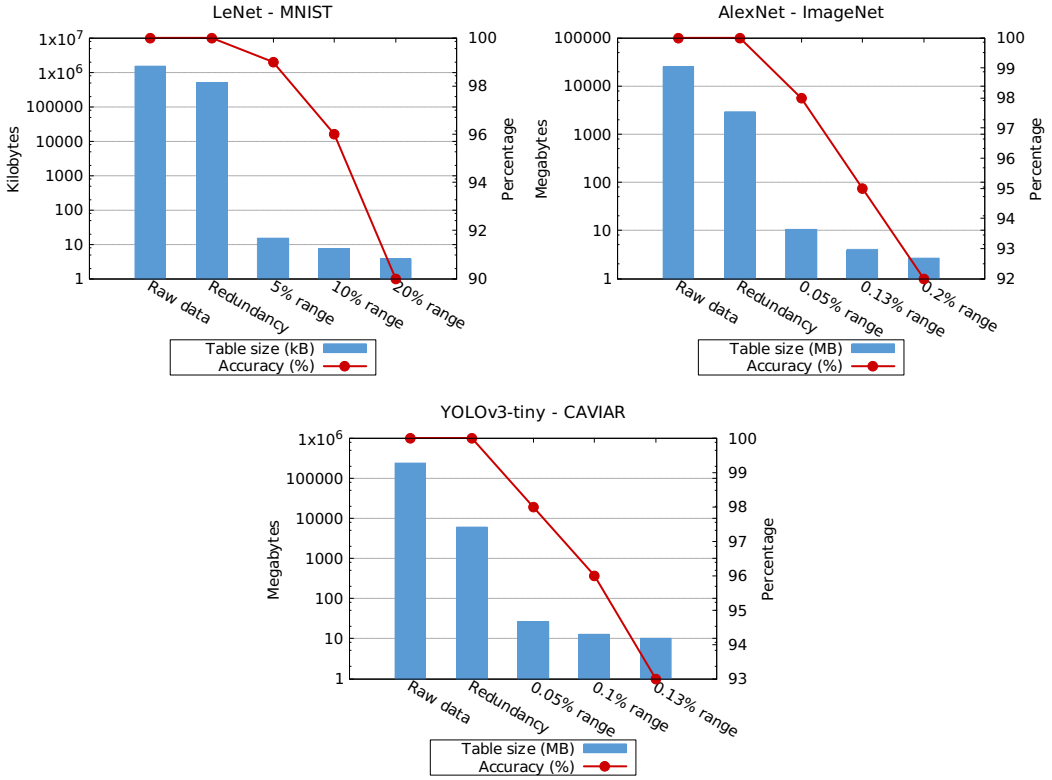
5.1.1 Lookup table sizes and accuracy

Figure 5.1 presents the lookup table sizes and their respective accuracy levels for profiling and running the three evaluation datasets above mentioned. On average, there is a slightly (in terms of magnitude) decrease in the demand for lookup table sizes for the MNIST dataset, ranging from 15 KB to 3.75 KB, with accuracy drops up to 10% only for the smallest table. This behavior happens because the MNIST dataset is composed of handwritten digits, which comprise small images containing static black backgrounds and white numbers, which allows the technique to perform a higher approximation without hurting the accuracy. The ImageNet dataset achieves similar accuracy levels to those presented for MNIST approximation. However, it demands bigger tables due to its variety present in the inputs and the more in deep CNN layout compared to LeNet. This dataset is comprised of worldwide images, which implies in various backgrounds representations. Thereby, this dataset requires a wider variety of numbers to represent a higher number of classes it comprises. Hence, the ImageNet does not allow greater levels of values approximation as those found in the MNIST, demanding table sizes varying from 10.5 MB to 2.62 MB, with an accuracy drop up to 8% for the most approximative scenario. Although the CAVIAR dataset is composed of surveillance camera videos, which may incur higher levels of data redundancy due to its static background, this dataset demands the biggest table sizes to achieve similar accuracy levels like those reported from the other networks execution, ranging from 27 MB to 10 MB. The higher demand for table sizes in CAVIAR can be explained due to the low accuracy value for the original CNN, so higher approximations cannot be done without incurring in more significant accuracy drops.

5.1.2 Inference time

To compare the convolution kernel-size reuse technique to a normal CNN execution, we took the average inference time for a single frame using the original CNN algorithm, and we normalized our results to the baseline scenario, as seen in Figure 5.2. The accelerations are strictly related to the depth of a CNN architecture, the input size, and mainly the targeted accuracy. First, small convolution kernel size and small networks, as found in the scenario LeNet-MNIST, reveals an improvement up to 1.9$\times$ for different setups of the clustering process. Though this achievement is not insignificant, it is much harder to beat the baseline in small kernel scenarios, where intermediate data resides

Figure 5.1: Table sizes and accuracy for the convolution kernel-size reuse technique



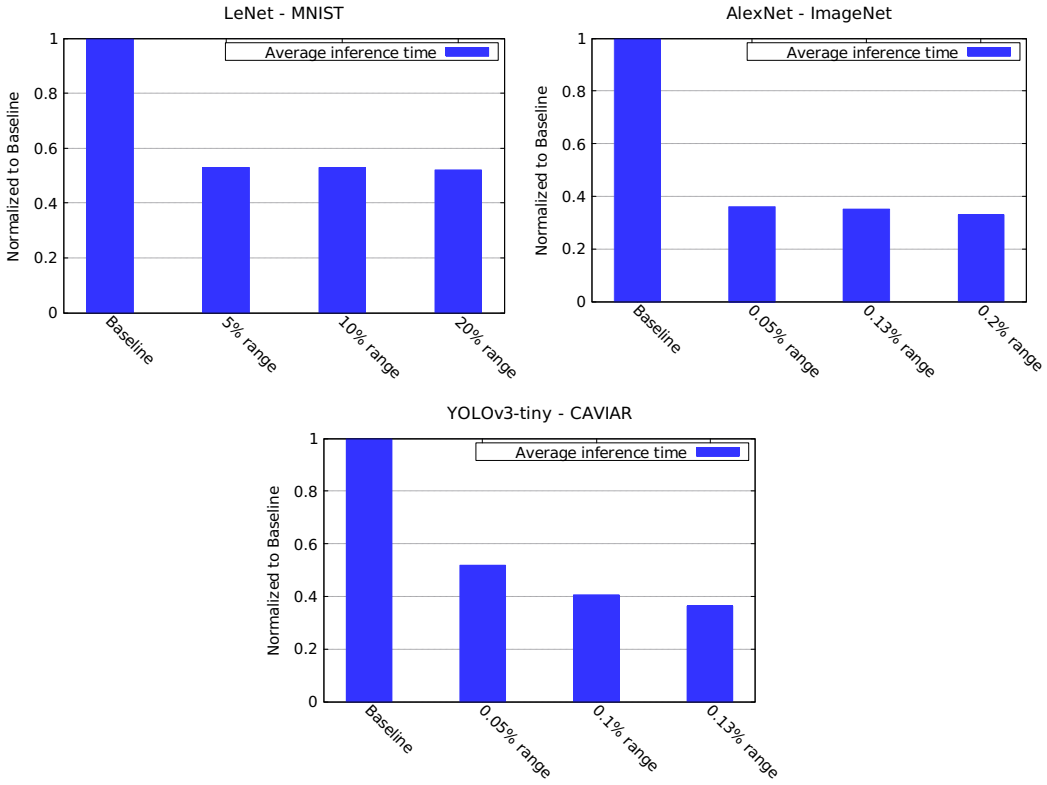
Source: Author

mostly on cache memories. The second observation relates accuracy, network size, and table size to the overall performance. The profiling tool has to use more classes due to the YOLO's depth to keep an acceptable level of accuracy, thus generating larger tables. In the execution time, searching on large tables increases the average memory access latency of our application, which limits the performance of the setup range 0.05% to 1.8× of the baseline. Nonetheless, when we go from the *RangeDistance* of 0.05% to 0.1%, there exists an accentuated slope in the average execution time chart. This effect can be explained due to the table size reduction (from 27 MB to 13 MB), where the lookup tables are likely to fit in the cache hierarchy of the processor described in Table 5.1. Finally, the small tables found in the profiling time for the AlexNet-Imagenet test-case reflect an average speed-up of 3× due to the medium size of this network.

5.1.3 Energy consumption

Figure 5.3 presents absolute values for DRAM transfer and the energy savings normalized to each baseline. From this figure, an important observation can be made: the energy savings are intrinsically related to the speedup and the off-chip data transfer be-

Figure 5.2: Average inference time comparison for the convolution kernel-size reuse technique

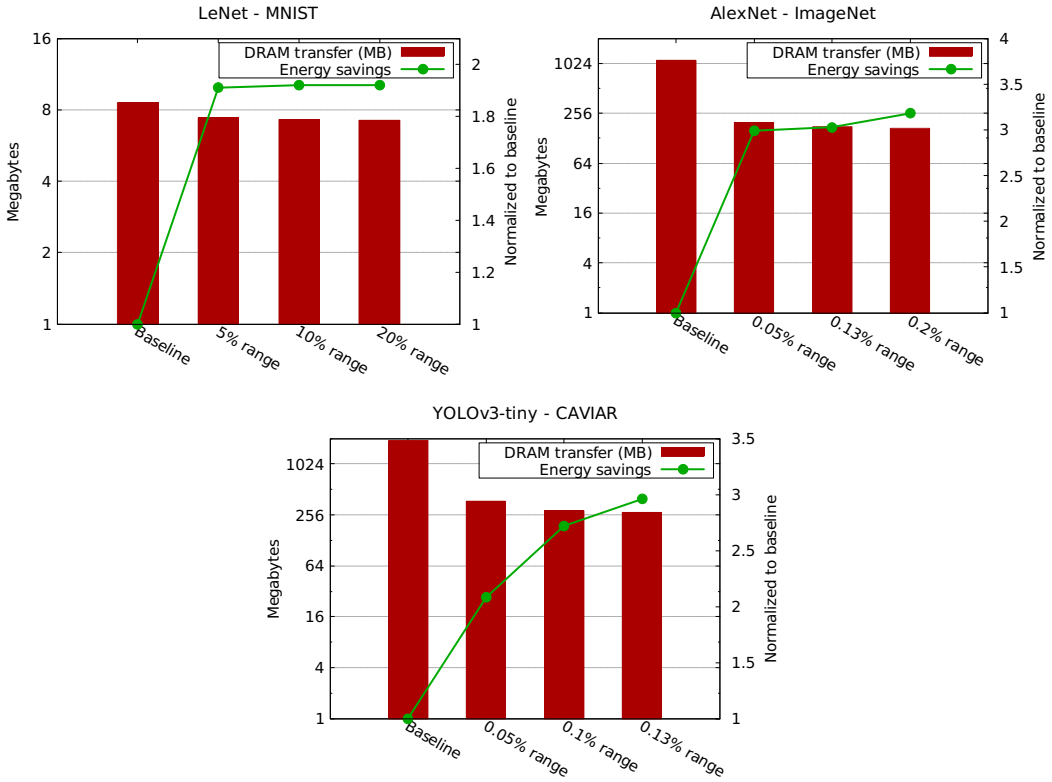


Source: Author

tween CPU and DRAM. Hence, the test-cases that require smaller tables will also require fewer DRAM accesses, which is likely to improve energy efficiency by a factor higher than the speedup of each scenario.

The LeNet-MNIST test-case outperforms the baseline by more than 1.9× by using the kernel-level reuse with a *RangeDistance* of 20%. Although the YOLOv3-tiny-CAVIAR test requires larger tables, it presents the most significant reduction in the off-chip transfer compared to the baseline. This behavior can be explained due to the presence of *inter-frame* locality, incurring a high degree of activation repetition due to the static background present in this dataset. Hence, the energy saving levels over the baseline can reach 2.9× using clustering configuration of "0.13% range". Finally, the AlexNet-ImageNet test-case can provide significant improvement in terms of energy, reaching up to 3.18× when compared to the baseline CNN execution. In conclusion, the data transfer results have significant contributions to overall energy savings: An execution of the computation reuse-based algorithm requires less data manipulation than the baseline convolution execution, which in turn reduces the data requests by the CPU. Alternatively, a more straightforward and less power-hungry CPU version could be used instead of the setup in Table 5.1. Furthermore, the clustering mechanism enables us to replace convolu-

Figure 5.3: Energy consumption comparison for the convolution kernel-size reuse technique



Source: Author

tion calculations by accesses to small lookup tables, leveraging data locality in the cache hierarchy, which in turn reduces the total DRAM transfers.

5.2 Grid-level reuse

To evaluate the convolution grid-level reuse technique presented in Section 4.2, we took the same setup employed in Section 5.1. Hence, the evaluated datasets were MNIST, ImageNet, and CAVIAR running on their respective CNN models. To perform comparisons regarding average inference time, accuracy, and energy savings, we generated all the grid-level tables based on the lookup tables created by the greatest clustering range distance employed for each dataset in Section 5.1. Thus, the *RangeDistance* values employed were 20%, 0.2%, and 0,13% for the datasets MNIST, ImageNet, and CAVIAR, respectivelly. Additionally, we built two evaluation scenarios for the grid-level reuse, gathering all the $2x2$, and $3x3$ grids in the profiling scenarios, and we selected those with a frequency higher than the average grid frequency value ($\#TotalGrids \div \#DifferentGrids$) of the profiling set to compose the tables. Table 5.2 presents all the evaluated scenarios, their

respective average grid frequency, and the average number of grids added to the lookup table corresponding to each convolution filter.

The average frequency of grids $2x2$ found in the entire profiling set (10,000 images) are 51,578; 421; and 19,406 for the respective test-cases: LeNet-MNIST, AlexNet-ImageNet, and YOLOv3-CAVIAR. The scenario LeNet-MNIST presents the highest average frequency and consequently needs fewer grids added to the lookup-tables (two on average). This elevated average grid frequency can be explained due to its simplicity in the inputs: only static black background images with white numbers. On the other hand, the ImageNet dataset has the most diverse set of representations among the tested scenarios. Thebery, the AlexNet-ImageNet scenario presents the worst result (lower average grid frequencies and higher added grids to lookup tables). Finally, due to its high inter-frame redundancy leveraged by the static background camera, the CAVIAR dataset also presents high levels of the grid frequency. Further, as we employ a bigger grid size (3×3), the average grid frequency values drop to 2,647; 43; and 1,145 for the three datasets, respectively. As a consequence of the grid-level reuse, the metric grid frequency will have a significant impact on accuracy, inference time, and energy savings, as will be presented in the following Sections.

Table 5.2: Grid-level reuse statistics

CNN/dataset	Grid-size	AVG grid freq.	AVG grids/table
LeNet-MNIST	2×2	51,578	2
	3×3	2,647	5
AlexNet-ImageNet	2×2	421	30,840
	3×3	43	97,530
YOLOv3-tiny-CAVIAR	2×2	19,406	152
	3×3	1,145	616

Source: Author

5.2.1 Lookup table sizes and accuracy

The table sizes and accuracy levels that enable grid-level reuse are shown in Figure 5.4. As the most frequent adjacent activations (i.e., bottom, right, and bottom-right activations for a $2x2$ grid) are stored along with a more precise activation, it is expected that the table size is increased by at least $4\times$ and $9\times$ for the $2x2$, and $3x3$ grid sizes, respectively. However, the table size depends on the threshold given in profiling time that determines the cutoff frequency for storing a pattern of the submatrices. Since the cutoff

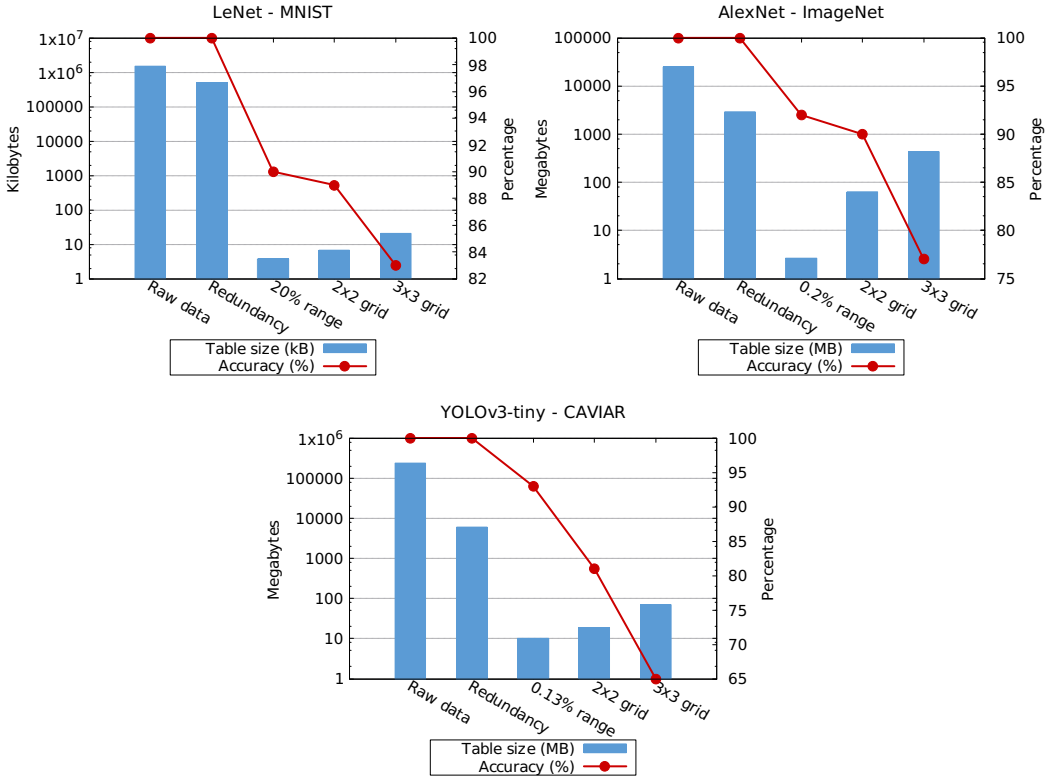
frequency in these experiments was given by the average number of repeated grids per image, the test-case scenarios present different overheads in table size. Thereby, as previously pointed out in Section 5.2, the average frequency of grids presented in Table 5.2 has a direct impact on the overhead in table sizes. Hence, AlexNet-Imagenet's tables are 24× and 164× larger than its kernel-level range clustering (0.2% range), demanding new table sizes of 62.76 MB and 430.59 MB for the two grid sizes evaluated. This result is strictly related to the diversity of images present in the dataset, where the recurrence of backgrounds is smaller compared to the other datasets. Meanwhile, YOLOv3-tiny-CAVIAR test-case has it tables increased by factors of 1.8× (10.2 MB to 18.47 MB) and 7× (10.42 MB to 70.34 MB), reinforcing the *inter-frame* locality found in sequential video frames. The LeNet-MNIST scenario presents the smallest table overhead (in the magnitude of values), demanding 3 KB (2.25× more space) and 16.87 KB (5.49× more space) of extra table size.

In addition to the table size, Figure 5.4 presents the accuracy levels for the evaluated grid-level reuse scenarios. The MNIST dataset presents the smaller reductions in the overall accuracy, from 90% in the 20% range to 89% in the $2x2$ grid scenario, and 83% in the $3x3$. The $3x3$ case has the highest accuracy drop level for the MNIST dataset (17%), indicating that the small convolution filter sizes (for the first convolution layer, the input size is 28×28) do not tolerate a grouping of convolutions in a matrix bigger than $2x2$. Despite the bigger table sizes required for the ImageNet scenario, its accuracy drops are relatively small for the $2x2$ grid size (10% of accuracy drop). Nonetheless, like the results for MNIST, when we employ a $3x3$ grid size, the accuracy drops grow up to 23%. Finally, the CAVIAR dataset presents the worst accuracy drops when employing the grid-level reuse: it presents accuracy drops of 19% and 35% for the two scenarios, respectively. Moreover, the worst-case scenario considers that the algorithm inserted errors due to speculative convolutions in the overlapping of activations, which were supposed to happen.

5.2.2 Inference time

When we add the grid-level to our computation reuse algorithm, one can observe the influence of dataset domain in the overall performance. Due to the predictive patterns in MNIST and CAVIAR datasets, we reduce the average inference time to 27-15% and 23%-20% of the baseline time, respectively. The small reduction in the CAVIAR dataset

Figure 5.4: Table sizes and accuracy for the convolution grid-size reuse technique

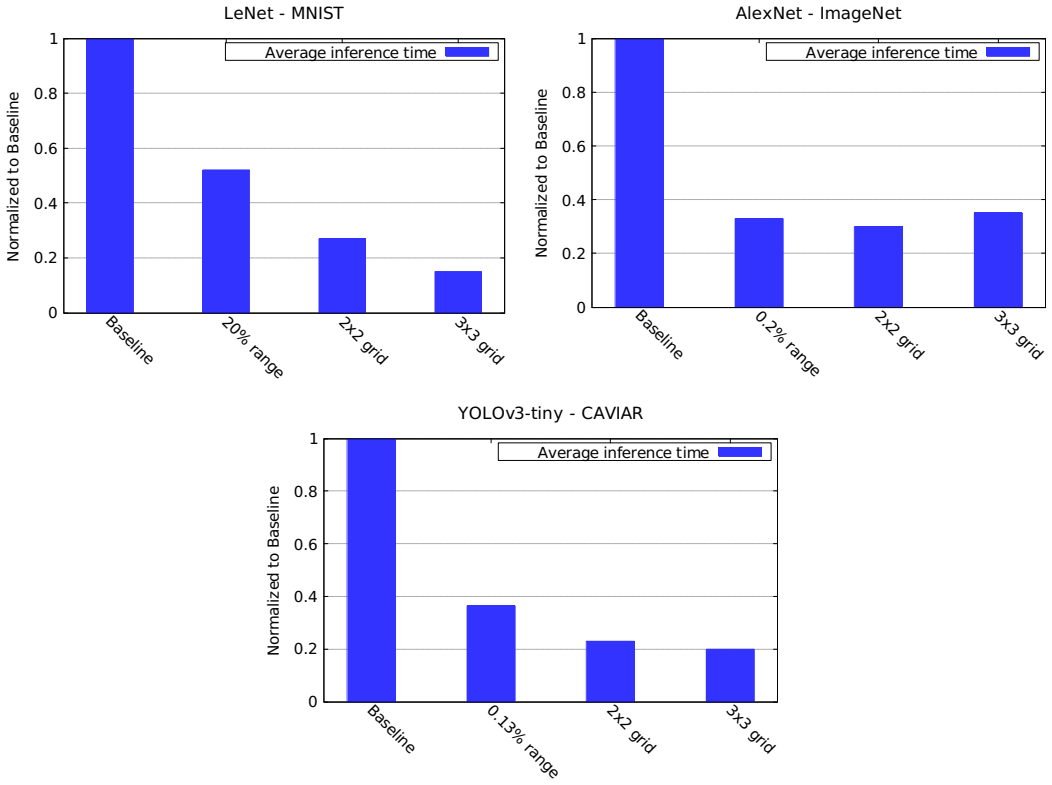


Source: Author

average inference time when we change from the $2x2$ scenario to the $3x3$ happens due to the table size increase ($7\times$) reported from the last. The table size overhead also has an impact on the inference time for the ImageNet dataset. For the $2x2$ grid size, the average inference time was reduced to 30% of the baseline, which configures a small increase when compared to the 33% achieved by the 0.2% range scenario. Furthermore, the results for a $3x3$ grid size get worse compared to the 0.2% range scenario, lowering the execution time to only 35% of the baseline.

Although the inference time for AlexNet-Imagenet test-case has worsened when compared to kernel-level reuse, this gives us an interesting sight: the deviation of image pixels in the dataset and the metrics related to the grid-level reuse in the profiling time are critical to deciding if the grid-level reuse must be used or not. The deviation of pixels in datasets impacts directly on the frequency of $3x3$ submatrices, thus providing more grids with lower reuse rate. Even if we change the current metric to gather fewer grids with the highest frequencies, which generates smaller tables, the reuse rate in the execution time is still reduced. Therefore, grid-level reuse may not be beneficial in a domain as varied as the Imagenet is.

Figure 5.5: Average inference time comparison for the convolution grid-size reuse technique



Source: Author

5.2.3 Energy consumption

Figure 5.6 presents absolute values for DRAM transfer and the energy normalized to each baseline when employing the convolution grid-level reuse technique. As already stated in Section 5.1.3, the DRAM transfer has a significant contribution to the overall energy savings. Furthermore, when employing the grid-level reuse, the off-chip transfers can be reduced even more, since not all the input are read due to the speculative execution of the grid-level reuse technique. The LeNet-MNIST test-case presents the best energy savings, outperforming the baseline scenario more than 3.73× by using a $2x2$ grid-size reuse, and by a factor of 6.8× regarding a $3x3$ grid-size. Regarding the DRAM transfer results for MNIST, the $2x2$ case reports a reduction from 8.6 MB to 6.5 MB, while the $3x3$ grid-size can reduce it to 6.3 MB. This slightly decrement in the DRAM transfer can be expected, since even the extra table size required for LeNet-MNIST (20.62 KB) is insignificant given cache memory sizes found in commercial CPUs (the CPU presented in Table 5.1 has 32 KB as the first-level of data cache).

The YOLOv3-tiny-CAVIAR scenario reaches energy savings of 4.6× and 5.2× compared to 2.96× related from the 0.13% range scenario. Notwithstanding the 1.8×

increasing in the table sizes required for the $2x2$ grid scenario, the acutest slope in the energy consumption curve can be seen when we go from the 0.13% range scenario to the $2x2$ grid size, outperforming the kernel-level reuse scenario in 1.55×, reducing the DRAM transfer from 288 MB to 279 MB. Moving from a $2x2$ to a $3x3$ grid introduces a table increase in 70× over the 0.13% range scenario. Thus, even with the expected reduction in the amount of data read (with a $3x3$ grid, we must read only two inputs to get results for nine outputs due to the speculation) when employed this technique, the DRAM transfer value reported reaches 330 MB.

Finally, the ImageNet dataset presents the worst energy savings results for the grid-level reuse technique. The $2x2$ case achieves energy savings of 3.42×, while the $3x3$ case lowers this value to 2.75× when compared to the 3.18× related from the 0.2% kernel-level reuse. Due to the extra lookup table size (60.14 MB and 427 MB) related from the grid-reuse applied to the ImageNet scenario, there is an increase in the DRAM transfer levels achieved by the grid-size scenarios (245 MB and 537 MB) compared to the 0.2% range (168 MB).

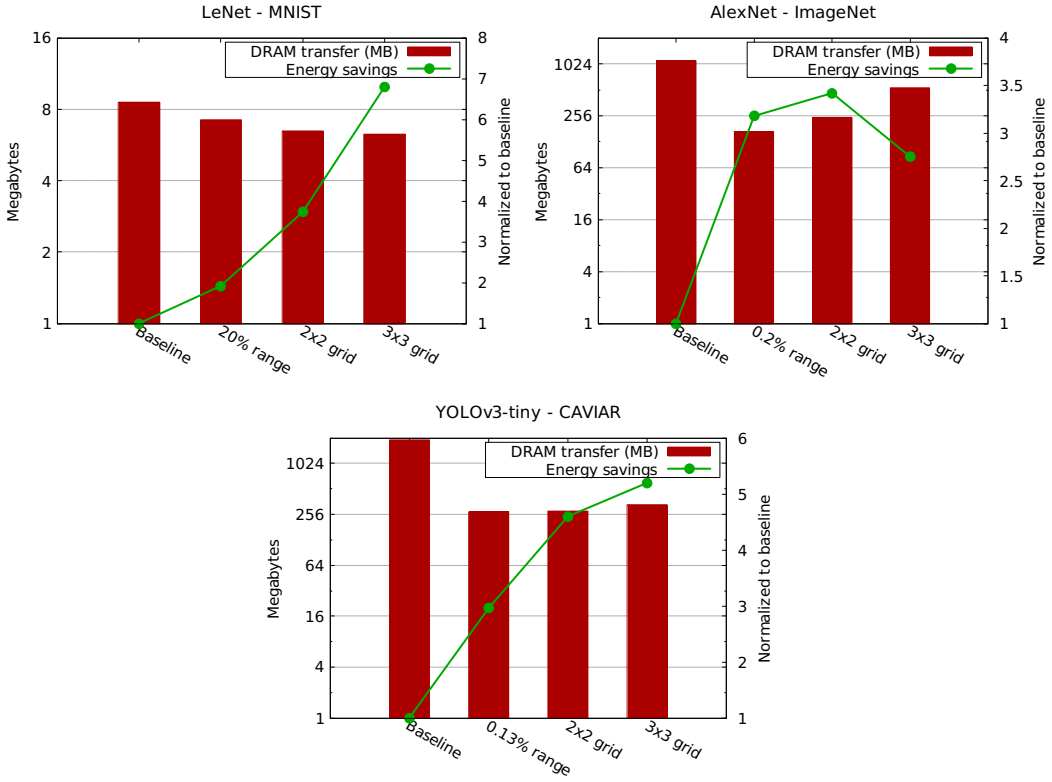
In conclusion, the extra lookup table sizes due to grid-level reuse have significant contributions to DRAM transfer results, and consequently to overall energy savings: When the new table sizes are much higher than the theoretical data transfer reduction (2× less reading operations for a $2x2$ grid, and 4.5× for a $3x3$ grid), the energy savings can be attenuated or even be smaller than the reported for the corresponding kernel-level reuse version.

5.3 Frame-level reuse

To evaluate the frame-level reuse technique proposed in Section 4.3, we selected the Datasets CAVIAR, Caltech's Pedestrian Detection benchmark, and GTEA. For evaluation, we selected 6,000 frames from each dataset. From CAVIAR, we selected videos of two different surveillance cameras, each recording a different corridor angle of a shopping center. From Caltech's dataset, we randomly select videos recorded by vehicles traveling in several traffic scenarios. From the GTEA dataset, we have chosen videos from different activities totaling the 6,000 required frames. The network used for our experiments was YOLOv3-tiny, which we will later replace with YOLOv3 to improve the work's scope in Section 5.3.4.

To better illustrate the impact of frame-level reuse on energy reduction, average

Figure 5.6: Energy consumption comparison for the convolution grid-size reuse technique



Source: Author

inference time, and accuracy, we have performed experiments using three different SAD thresholds for each dataset. These values were obtained by evaluating block matching results from the training set. First, we save the maximum SAD found between each two frames comparison. We can then calculate a threshold for each dataset based on this maximum value, setting it to be a certain percentage from the maximum value. The higher the threshold, the more differences between images are tolerated, and so the computation reuse method is called fewer times. As such, applications that do not tolerate much error need smaller thresholds. We have chosen thresholds of 50%, 70%, and 85%, to exemplify the behavior of our method for different rates of re-computation.

Regarding the "Computation reuse" box presented in the execution flow that describes this technique in Figure 4.11, we replaced the CNN execution calls by the convolution kernel-level computation reuse, with the same setup employed for the CAVIAR dataset in Section 5.1. As the *RangeDistance*, we took the scenario 0.13% since it can achieve the best speedup presented over the baseline. As the input training set for the *Computation Reuse profiler* execution, we used a batch size of 10,000 video frames for each dataset, different from those 6,000 used for evaluation. The choice for employing the kernel-level reuse instead of grid-level was based on the accuracy results for the CAVIAR dataset illustrated in Section 5.1.1. Moreover, given the same CNN layout employed in

this experiment, and the resemblance present in the characteristics of the datasets, we kept the same 0.13% as the *RangeDistance* for the three evaluated datasets in this section.

5.3.1 Accuracy

Table 5.3 illustrates the relationship between inference accuracy levels and CNN call rate, when running a pre-trained YOLOv3-tiny model on the validation datasets. For all the datasets, it can be seen that as the number of frames that need to be inferred with our computation reuse technique decreases, so does the accuracy. Further, one can notice that for the CAVIAR dataset, higher levels of accuracy can be achieved with less CNN execution calls, while Caltech demands more elevated CNN execution rate. This *accuracy versus CNN execution rate* behavior can be explained due to the datasets video classes: the CAVIAR dataset presents higher levels of locality since the source camera resides in a fixed observation point. The GTEA dataset presents behavior that falls in the middle of these, as the 50% threshold is probably too low, making it so that too many frames need re-computation, which leads to higher execution rate. As the threshold lowers, however, we find that it gets closer to results from the CAVIAR dataset. This is expected, as the GTEA dataset, while not made from a fixed camera, does not have as many angles and background variations as the Caltech one.

Table 5.3: YOLOv3-tiny CNN accuracy and inferred frames

Dataset	Scenario	CNN call(%)	Accuracy(%)
Caltech	Baseline	100	100
	Frame prediction threshold 50%	61	95
	Frame prediction threshold 70%	44	92
	Frame prediction threshold 85%	31	84
CAVIAR	Baseline	100	100
	Frame prediction threshold 50%	55	98
	Frame prediction threshold 70%	38	94
	Frame prediction threshold 85%	23	87
GTEA	Baseline	100	100
	Frame prediction threshold 50%	86	98
	Frame prediction threshold 70%	31	93
	Frame prediction threshold 85%	16	83

Source: Author

Furthermore, three significant sources of error can influence the output, either resulting in non-detected objects or mispredictions:

1. First, the computation reuse technique employed performs approximations in the

output, to reduce the lookup table sizes further. As such, higher rates of CNN executions using computation reuse are expected the accuracy levels to get closer to those related in Section 5.1.1.

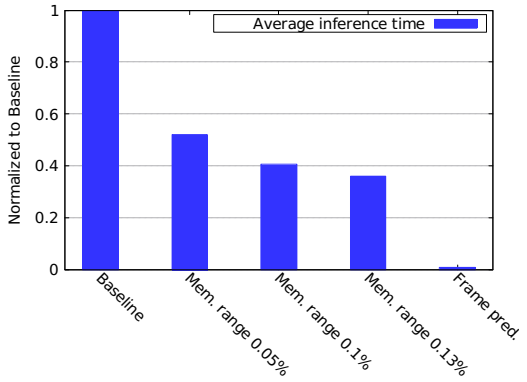
2. The motion prediction algorithm can introduce inaccuracy levels in the inference through wrong estimations of the bounding boxes when the frames change quickly.
3. Furthermore, if the CNN execution erroneously makes a prediction mistake, then it is possible that the motion estimation technique will propagate the error. For example, if the network finds an object where there is none, the motion estimation will keep this object's prediction until another CNN call is made. Similarly, if the network fails to find an object, then our technique will not be able to detect this object until at least the next CNN inference.

5.3.2 Inference time

To estimate the speedup achieved by our proposed technique over a normal CNN execution, we take a single frame inference time by the CNN and normalize our results to the baseline scenario, as seen in Figure 5.7. All the *Mem. range* columns refer to the Computation Reuse technique, and represent the clustering range distance used to generate the lookup tables. Thus, a 0.05% range means that numbers were grouped in clusters with a distance value set to 0.05% of their original values. The range choice of 0.05% manages to perform a frame inference 1.92× faster than the baseline. Further, increasing the distance outcomes a smaller table, which in turn introduces accuracy drops but accelerates the frame inference. As such, a 0.1% range provides 2.46× faster inference. Finally, regarding range the value of 0.13%, one can reach inference time accelerations up to 2.73×.

The *Frame pred.* column refers to the time needed by the motion estimation technique to make a frame inference. It is 116× faster than the baseline time. However, we cannot perform frame predictions only: at least one CNN execution inference as the "cold start state" to completely compute the motion estimations is required. On top of that, too many predictions in a row can lead to accuracy drops. By combining the Computation Reuse and the frame prediction techniques, however, we can exploit the inference time gap between the original inference time and our achieved speedups. If the FPS rate of the original video is kept the same, significant and reduced energy consumption can be

Figure 5.7: Average frame inference time comparison over the baseline system



Source: Author

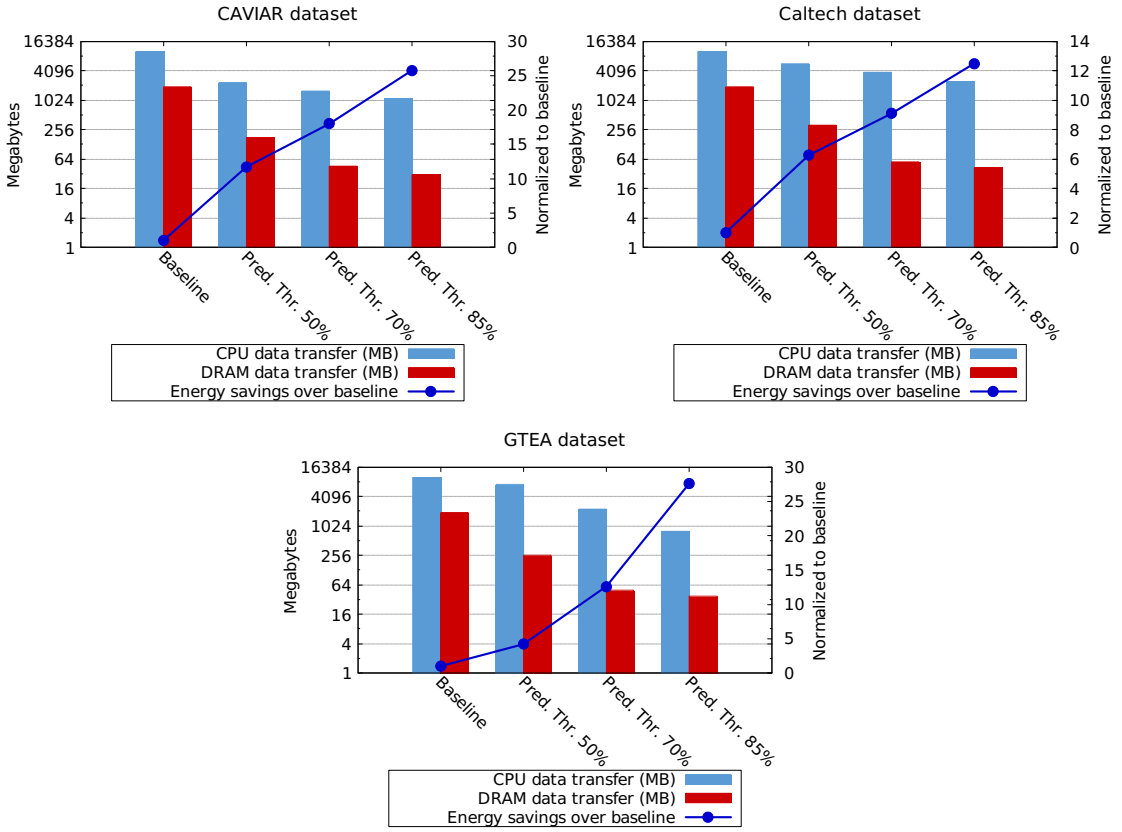
obtained, as will be presented in Section 5.3.3.

5.3.3 Energy consumption

As already presented, one can achieve speedups of up to 2.73× and 116× when replacing CNN inference with computation reuse and motion prediction, respectively. As such, those two techniques combined can preserve FPS rates while drastically improving energy gains. To do so, a mechanism such as Dynamic Voltage and Frequency Scaling (DVFS) can be used to reduce the main CPU frequency operation, lowering its power dissipation while accomplishing the target inference time for each frame. Alternatively, a simpler and less power-hungry CPU version could be used in place of the original. Figure 5.8 illustrates the energy savings that one can achieve in comparison to the baseline, which can be done just by reducing the frequency operation of the baseline system.

For motion prediction, we use the three chosen thresholds and demonstrate how they impact the total energy consumption. Higher thresholds imply in more frames going through motion prediction instead of computation reuse inference, which makes energy gains more pronounced, since the latter achieves small energy savings compared to the former. As a tradeoff, more significant energy savings impact more on the accuracy, as shown in Section 5.3.1. The most significant energy gains can be seen in the GTEA dataset with a threshold of 85%, where our technique can manage energy savings of up to 27.62× over the baseline. A threshold of 70% yields 12.61×, while 50% can achieve 4.20×. For the Caltech dataset, the thresholds of 85%, 70%, and 50% produce energy gains of 12.5, 9.09, and 6.25 times, respectively. As for CAVIAR, the same thresholds obtain energy gains of 11.70, 17.97 and 25.77. The data pattern can easily explain the dif-

Figure 5.8: Energy gains over baseline and data transfers running three different datasets on YOLOv3-tiny.



Source: Author

ference in gains between the three datasets: as mentioned, CAVIAR is composed of fixed camera videos, which our technique can benefit more from. On the other hand, Caltech's dataset has regularly moving car cameras, presenting less opportunity for motion prediction, as frames change more frequently. GTEA presents behavior that's in the middle of them, with a moving camera that is slower than a camera in a car and reduced fluctuations between the other two. While it can achieve more energy gains than CAVIAR with an 85% threshold, those gains diminish with a 50% threshold, with which CAVIAR achieves 11.70× energy reduction, while GTEA only manages 4.20×. Still, a factor of more than 12× of energy reduction in software can certainly be useful.

Furthermore, the *"CPU data transfer"* bars present the amount of data generated and requested by the CPU to the first level of cache memory. The *"CPU data transfer"* value amounts to approximately 10 GB of data for the baseline system running the three evaluation sets. When applying our technique, considering the three chosen thresholds, the data transfers are reduced to ranges of 2316-7135 MB, 1540-3808 MB, and 808-2469 MB. Meanwhile, the *"DRAM data transfer"* bars refer to the amount of data transferred between the DRAM module and the cache hierarchy. Similarly, the *"DRAM*

data transfer" series reach approximately 1892 MB of data for the baseline system running the three evaluation sets. For the three chosen thresholds, they are reduced to ranges of 175-312 MB, 45-55 MB, and 31-42 MB.

In conclusion, the data transfer results have significant contributions to overall energy savings: An execution of the motion prediction algorithm requires less data manipulation than a baseline convolution execution, which in turn reduces the data requests by the CPU and the DRAM accesses. Furthermore, the clustering mechanism of the computation reuse technique replaces convolution calculations by accesses to small lookup tables, leveraging data locality in the cache hierarchy, which in turn reduces the total DRAM transfers. Last but not least, one can see that the CAVIAR and GTEA datasets demand even fewer data management efforts than the Caltech's one, keeping in mind the frame locality behavior related in the previous results.

5.3.4 Applying the technique to a deeper NN

To expand the representativeness and generality of this work, we conducted several experiments with a deeper YOLOv3 variation. As this network is much slower to run than YOLOv3-tiny, we were not able to gather complete data as described before, but we will exemplify that our technique could be expanded to models other than YOLOv3-tiny as well. DNN have reported more significant accuracy levels since they can extract more features of the inputs along with their more numerously convolutional layers. Table 5.4 presents a few versions of the YOLOv3's original layout. One can notice that the YOLOv3-tiny variation comprises of the smallest, the less computation power dependent, and also the less accurate model. On the other hand, the YOLOv3 version presents a better ratio between accuracy and computation demand. It comprises up to 75 convolutional layers, and delivers a mAP score for the MSCOCO dataset (LIN et al., 2014) of 55.3%.

Using YOLOv3-tiny, our evaluations have presented a 16% error (84% accuracy), caused by situations explained in Section 5.3.1. As shown in Table 5.4, this tiny network achieves a mAP score of only 33.1% on the MSCOCO dataset, compared to YOLOv3's score of 55.3%. Thereby, both to demonstrate the applicability of the proposed work, and to evaluate the possibility that some accuracy drops reported in the previous results could be fixed only by changing our network choice to a more reliable one; we employed the YOLOv3 variation to conduct some experiments.

Following this logic, in this section, we present some results for the CAVIAR

dataset, when running it on the YOLOv3 as the baseline network model. As seen in Figure 5.9, for the same threshold of 85%, the less beneficial for accuracy, applying the same technique yields an accuracy of 98%, compared to YOLOv3-tiny's 87%. These results can be easily explained: YOLOv3 is less likely to make mispredictions and is also less likely to entirely miss objects in a frame, problems that were previously described in Section 5.3.1. Thus, the error propagated by the motion prediction technique is also reduced. Furthermore, increasing the threshold to 99% in this network model yields an accuracy of 97%, which is only a 1% difference in comparison to the 85% threshold.

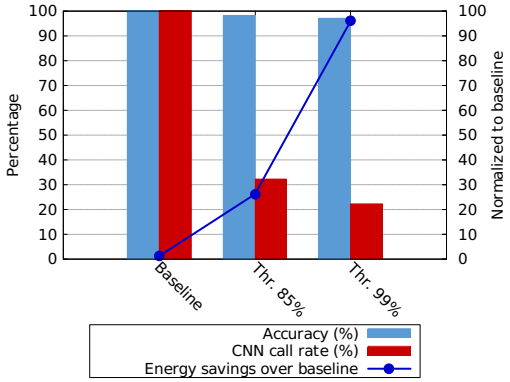
Table 5.4: YOLOv3 variations comparison

	YOLOv3-tiny	YOLOv3	YOLOv3-spp
Total layers	25	108	115
Conv. layers	13	75	76
GFLOPS	5.56	65.86	141.45
MSCOCO mAP(%)	33.1	55.3	60.6

Source: Author

Regarding energy reduction, gains are also more aggressive because we can further limit frames that will need to be processed by the computation reuse technique. From the 26 times energy savings achieved by the 85% threshold, we can go to savings of up to 96 times, while reducing accuracy only by 1%. These saving can be explained by the full inference call rate, which drops from 32% with an 85% threshold to 22% with a 99% threshold. In other words, the more complex and precise the network, the smaller our errors, and the larger the possible acceleration provided by frame skipping.

Figure 5.9: Energy gains over baseline of the CAVIAR dataset using YOLOv3.



Source: Author

6 CONCLUSION AND FUTURE WORK

In this work, we analyzed the execution of different CNN models and conducted experiments to expose computation reuse opportunities in these networks. Based on input recurrence patterns at convolution kernel-level, we exploited data redundancy and similarity by using a software-based reuse technique to skip entire convolution operations by keeping representative activations in tables. We designed a tool that analyzes a profiling set and generates hash tables for future reuse in the running time. By using a range-based clustering, we made the size of the lookup tables smaller at a cost to the accuracy of the prediction. Further, we extended the reuse level to a grid of convolutions by storing neighboring convolutions values. Through the speculative execution of the adjacent convolutions, it was possible to reduce the input read operations by theoretical factors of $2\times$ and $4.5\times$, for a $2x2$ and a $3x3$ convolution grid-size, respectively. Finally, we presented a combination of two software-only strategies to reduce the energy consumption of CNN video inference. The proposed technique could exploit the intra-frame and the inter-frame locality inherent from sequential frames, relying on a computation reuse technique and on video compression basics to predict frames.

Our experimental results show that our implementation can reduce the average inference time up $3\times$ for a kernel-level reuse granularity, $6.6\times$ for a $3x3$ grid-level, and $116\times$ for the motion prediction technique. We have shown how the performance and energy improvements provided by our techniques are strictly dependent on the table sizes required for the computation reuse technique, and the DRAM transfer levels. Thus, the most aggressive energy savings can be improved by a factor of $3.18\times$ for the kernel-level reuse, $6.6\times$ for a $3x3$ grid-level, and $27.62\times$ for the frame-level reuse. Additionally, as we either perform approximations to reduce the lookup tables or predict frame inferences, there are inserted accuracy drops in the results. The accuracy level can be a limiting factor to determine whether each granularity of reuse is suitable for each CNN and dataset domain pairs. For LeNet-MNIST case, the highest speedup for the kernel-level reuse introduces an accuracy drop of 10% and 11% for the $2x2$ grid size scenario. The AlexNet-ImageNet relates accuracy drops up to 5% and 10% for the same scenarios, respectively. However, the YOLOv3-tiny-CAVIAR presents accuracy drops of 7% in the kernel-level, and 19% in the $2x2$ grid-level, revealing the grid-level technique not feasible for this specific dataset-CNN pair.

Bringing the table presented in Section 3.3 and comparing with the features cov-

ered by this work, according to the characteristics listed in Table 6.1, this work is the only one which provides a reduction in the memory data transfers, reduces the total number of multiplications, convolutions, and CNN calls inferences, while not demanding to retrain the original CNN neither a dedicated hardware accelerator.

Table 6.1: Comparison between this work and related work

	ASIC	Less memory transfers	Less #Mult	Less #Conv	Less #Inferences	Retraining
Chen et al. (2014a)	✓	✓	✓	✗	✗	✗
Chen et al. (2017)	✓	✓	✓	✗	✗	✗
Riera et al. (2018)	✓	✓	✓	✗	✗	✗
Hegde et al. (2018)	✓	✓	✓	✗	✗	✗
Raha et al. (2017)	✓	✓	✓	✗	✗	✗
Razlighi et al. (2017)	✓	✓	✓	✗	✗	✗
Jiao et al. (2018)	✓	✗	✓	✗	✗	✗
Morecino et al. (2019)	✓	✗	✓	✗	✗	✗
Zhu et al. (2018)	✓	✓	✗	✗	✓	✗
Hubara et al. (2017)	✗	✓	✓	✗	✗	✗
Han et al. (2015)	✗	✓	✓	✗	✗	✗
Wu et al. (2016)	✗	✓	✗	✗	✗	✗
Liu et al. (2018)	✗	✓	✓	✗	✗	✗
Howard et al. (2017)	✗	✓	✓	✓	✗	✓
Shafiee et al. (2017)	✗	✓	✓	✓	✗	✓
Kang et al. (2017)	✗	✓	✓	✓	✗	✓
Ji et al. (2013)	✗	✓	✗	✗	✓	✗
This work	✗	✓	✓	✓	✓	✗

Source: Author

As future work, we intend to employ scratchpads to store the lookup tables used for reuse more effectively, thus achieving even more significant energy savings. Also, the exploitation of bitwise operations and parallel memory arrays found in Field-Programmable Gate Array (FPGA) devices, could be advantageous for a hardware implementation of this work. Further, we also plan on testing our technique with more network models, such as gathering full results for more in-depth versions of YOLOv3 CNN and extending it to run on GPUs.

6.1 Published Papers

During the master degree, this work culminated in a published paper, in the International Conference on Embedded Computer Systems, and other publications not related

to the current thesis, as listed below:

1. DE MOURA, Rafael Fao et al. **Skipping CNN convolutions through efficient memoization**. In: 2019 International Conference on Embedded Computer Systems: Architectures, MOdeling and Simulation (SAMOS). IEEE, 2019.
2. SANTOS, Paulo Cesar et al. **A Technologically Agnostic Framework for Cyber-Physical and IoT Processing-in-Memory-based Systems Simulation**. In: Microprocessors and Microsystems, 2019.
3. AHMED, Hameeza et al. **A Compiler for Automatic Selection of Suitable Processing-in-Memory Instructions**. In Proceedings of the Conference on Design, Automation & Test in Europe European Design and Automation Association. IEEE, 2019.
4. DE LIMA, João Paulo C. et al. **Exploiting Reconfigurable Vector Processing for Energy-Efficient Computation in 3D-Stacked Memories**. In: International Symposium on Applied Reconfigurable Computing. Springer, Cham, 2019. p. 262-276.
5. SANTOS, Paulo Cesar et al. **Exploring IoT platform with technologically agnostic processing-in-memory framework**. In: Proceedings of the Workshop on INTelligent Embedded Systems Architectures and Applications. ACM, 2018. p. 1-6.

REFERENCES

AZMAT, M.; SCHUHMAYER, C. Future scenario: Self driving cars-the future has already begun. In: **Conference paper, DOI**. [S.l.: s.n.], 2015. v. 10.

CHEN, T. et al. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In: ACM. **ACM Sigplan Notices**. [S.l.], 2014. v. 49, n. 4, p. 269–284.

CHEN, Y. et al. Dadiannao: A machine-learning supercomputer. In: IEEE COMPUTER SOCIETY. **Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture**. [S.l.], 2014. p. 609–622.

CHEN, Y.-H. et al. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. **IEEE Journal of Solid-State Circuits**, IEEE, v. 52, n. 1, p. 127–138, 2017.

CHEN, Y.-S.; HUNG, Y.-P.; FUH, C.-S. Fast block matching algorithm based on the winner-update strategy. **IEEE Transactions on Image Processing**, IEEE, v. 10, n. 8, p. 1212–1222, 2001.

CHOQUETTE, J.; GIROUX, O.; FOLEY, D. Volta: performance and programmability. **IEEE Micro**, IEEE, v. 38, n. 2, p. 42–52, 2018.

DENG, J. et al. Imagenet: A large-scale hierarchical image database. In: IEEE. **2009 IEEE conference on computer vision and pattern recognition**. [S.l.], 2009. p. 248–255.

DENG, L. The mnist database of handwritten digit images for machine learning research [best of the web]. **IEEE Signal Processing Magazine**, IEEE, v. 29, n. 6, p. 141–142, 2012.

DOLLÁR, P. et al. Pedestrian detection: A benchmark. IEEE, 2009.

FATHI, A.; REN, X.; REHG, J. M. Learning to recognize objects in egocentric activities. In: IEEE. **CVPR 2011**. [S.l.], 2011. p. 3281–3288.

FISHER, R.; SANTOS-VICTOR, J.; CROWLEY, J. Caviar test case scenarios. **URL http://homepages. inf. ed. ac. uk/rbf/CAVIAR**, 2004.

GONG, Y. et al. Compressing deep convolutional networks using vector quantization. **arXiv preprint arXiv:1412.6115**, 2014.

GOODFELLOW, I. et al. Generative adversarial nets. In: **Advances in neural information processing systems**. [S.l.: s.n.], 2014. p. 2672–2680.

HAN, S.; MAO, H.; DALLY, W. J. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. **arXiv preprint arXiv:1510.00149**, 2015.

HEGDE, K. et al. Ucnn: Exploiting computational reuse in deep neural networks via weight repetition. In: IEEE PRESS. **Proceedings of the 45th Annual International Symposium on Computer Architecture**. [S.l.], 2018. p. 674–687.

HOWARD, A. G. et al. Mobilenets: Efficient convolutional neural networks for mobile vision applications. **arXiv preprint arXiv:1704.04861**, 2017.

HUBARA, I. et al. Quantized neural networks: Training neural networks with low precision weights and activations. **The Journal of Machine Learning Research**, JMLR. org, v. 18, n. 1, p. 6869–6898, 2017.

JAKUBOWSKI, M.; PASTUSZAK, G. Block-based motion estimation algorithms—a survey. **Opto-Electronics Review**, Springer, v. 21, n. 1, p. 86–102, 2013.

JI, S. et al. 3d convolutional neural networks for human action recognition. **IEEE transactions on pattern analysis and machine intelligence**, IEEE, v. 35, n. 1, p. 221–231, 2013.

JIAO, X. et al. Energy-efficient neural networks using approximate computation reuse. In: IEEE. **2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)**. [S.l.], 2018. p. 1223–1228.

KANG, D. et al. Noscope: optimizing neural network queries over video at scale. **Proceedings of the VLDB Endowment**, VLDB Endowment, v. 10, n. 11, p. 1586–1597, 2017.

KARPATHY, A. et al. Large-scale video classification with convolutional neural networks. In: **Proceedings of the IEEE conference on Computer Vision and Pattern Recognition**. [S.l.: s.n.], 2014. p. 1725–1732.

KRIZHEVSKY, A.; SUTSKEVER, I.; HINTON, G. E. Imagenet classification with deep convolutional neural networks. In: **Advances in neural information processing systems**. [S.l.: s.n.], 2012. p. 1097–1105.

LECUN, Y. et al. Handwritten digit recognition with a back-propagation network. In: **Advances in neural information processing systems**. [S.l.: s.n.], 1990. p. 396–404.

LECUN, Y. et al. Lenet-5, convolutional neural networks. **URL: http://yann. lecun. com/exdb/lenet**, v. 20, 2015.

LI, Y.; YE, Z.; REHG, J. M. Delving into egocentric actions. In: **Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition**. [S.l.: s.n.], 2015. p. 287–295.

LIN, T.-Y. et al. Microsoft coco: Common objects in context. In: SPRINGER. **European conference on computer vision**. [S.l.], 2014. p. 740–755.

LIU, W. et al. A survey of deep neural network architectures and their applications. **Neurocomputing**, Elsevier, v. 234, p. 11–26, 2017.

LIU, X. et al. Efficient sparse-winograd convolutional neural networks. **International Conference on Learning Representations (ICLR)**, 2018.

MOCERINO, L.; TENACE, V.; CALIMERA, A. Energy-efficient convolutional neural networks via recurrent data reuse. In: IEEE. **2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)**. [S.l.], 2019. p. 848–853.

MURALIMANOHAR, N.; BALASUBRAMONIAN, R.; JOUPPI, N. P. Cacti 6.0: A tool to model large caches. **HP laboratories**, p. 22–31, 2009.

NG, J. Y.-H. et al. Beyond short snippets: Deep networks for video classification. In: **Proceedings of the IEEE conference on computer vision and pattern recognition**. [S.l.: s.n.], 2015. p. 4694–4702.

RAHA, A.; RAGHUNATHAN, V. q lut: Input-aware quantized table lookup for energy-efficient approximate accelerators. **ACM Transactions on Embedded Computing Systems (TECS)**, ACM, v. 16, n. 5s, p. 130, 2017.

RAZLIGHI, M. S. et al. Looknn: Neural network with no multiplication. In: EUROPEAN DESIGN AND AUTOMATION ASSOCIATION. **Proceedings of the Conference on Design, Automation & Test in Europe**. [S.l.], 2017. p. 1779–1784.

REDMON, J. **Darknet: Open Source Neural Networks in C**. 2013–2016. <http://pjreddie.com/darknet/>.

REDMON, J.; FARHADI, A. Yolo9000: better, faster, stronger. In: **Proceedings of the IEEE conference on computer vision and pattern recognition**. [S.l.: s.n.], 2017. p. 7263–7271.

REDMON, J.; FARHADI, A. Yolov3: An incremental improvement. **arXiv preprint arXiv:1804.02767**, 2018.

RIERA, M.; ARNAU, J.-M.; GONZÁLEZ, A. Computation reuse in dnns by exploiting input similarity. In: IEEE PRESS. **Proceedings of the 45th Annual International Symposium on Computer Architecture**. [S.l.], 2018. p. 57–68.

ROSENBLATT, F. The perceptron: a probabilistic model for information storage and organization in the brain. **Psychological review**, American Psychological Association, v. 65, n. 6, p. 386, 1958.

SHAFIEE, M. J. et al. Fast yolo: a fast you only look once system for real-time embedded object detection in video. **arXiv preprint arXiv:1709.05943**, 2017.

SODANI, A. Knights landing (knl): 2nd generation intel® xeon phi processor. In: IEEE. **2015 IEEE Hot Chips 27 Symposium (HCS)**. [S.l.], 2015. p. 1–24.

SPRINGENBERG, J. T. et al. Striving for simplicity: The all convolutional net. **arXiv preprint arXiv:1412.6806**, 2014.

SZEGEDY, C. et al. Inception-v4, inception-resnet and the impact of residual connections on learning. In: **Thirty-First AAAI Conference on Artificial Intelligence**. [S.l.: s.n.], 2017.

SZEGEDY, C. et al. Going deeper with convolutions. In: **Proceedings of the IEEE conference on computer vision and pattern recognition**. [S.l.: s.n.], 2015. p. 1–9.

WU, J. et al. Quantized convolutional neural networks for mobile devices. In: **Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition**. [S.l.: s.n.], 2016. p. 4820–4828.

ZEILER, M. D.; FERGUS, R. Visualizing and understanding convolutional networks. In: SPRINGER. **European conference on computer vision**. [S.l.], 2014. p. 818–833.

ZHU, Y. et al. Euphrates: Algorithm-soc co-design for low-power mobile continuous vision. **arXiv preprint arXiv:1803.11232**, 2018.

ZOU, K. et al. Learn-to-scale: Parallelizing deep learning inference on chip multiprocessor architecture. In: IEEE. **2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)**. [S.l.], 2019. p. 1172–1177.