# The Best of Many Worlds: Efficient Machine Learning Inference on Heterogeneous Hardware Architectures

*Rafael Tsirbas*

Thesis submitted in partial fulfillment of the requirements for the

*Masters' of Science degree in Computer Science and Engineering*

University of Crete
School of Sciences and Engineering
Computer Science Department
Voutes University Campus, 700 13 Heraklion, Crete, Greece

Thesis Advisors: Prof. *Evangelos Markatos*, Dr. *Sotiris Ioannidis*

This work has been performed at the University of Crete, School of Sciences and Engineering, Computer Science Department.

The work has been supported by the Foundation for Research and Technology - Hellas (FORTH), Institute of Computer Science (ICS).

University of Crete
Computer Science Department

**The Best of Many Worlds: Efficient Machine Learning Inference on Heterogeneous Hardware Architectures**

Thesis submitted by
**Rafael Tsirbas**
in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

THESIS APPROVAL

Author: _____
Rafael Tsirbas

Committee approvals: _____
Evangelos Markatos
Professor, Thesis Supervisor

_____
Sotiris Ioannidis
Professor, Thesis Supervisor

_____
Polyvios Pratikakis
Assistant Professor, Committee Member

Departmental approval: _____
Polyvios Pratikakis
Assistant Professor, Director of Graduate Studies

Heraklion, September 2020

# The Best of Many Worlds: Efficient Machine Learning Inference on Heterogeneous Hardware Architectures


## Abstract

Heterogeneous and asymmetric computing systems are composed by a set of different processing units, each with its own unique performance and energy characteristics. Still, the majority of current machine learning applications targets only a single device (the CPU or some accelerator), leaving the rest processing resources unused and idle. In this work, we propose an adaptive scheduling approach that supports heterogeneous and asymmetric hardware, tailored for a diversified set of machine learning models. Our scheduler can respond quickly to dynamic performance fluctuations that occur at real-time, such as data bursts, application overloads and system changes. The experimental results show that it is able to match the peak throughput of a diverse set of machine learning models, by predicting correctly the appropriate device with an accuracy of 92.5%, while consuming up to 10% less energy.

## Περίληψη

Τα ετερογενή υπολογιστικά συστήματα απαρτίζονται από ένα σύνολο υπολογιστικών συσκευών, κάθε μια από τις οποίες έχει τα δικά της χαρακτηριστικά κατανάλωσης ενέργειας καθώς και την απόδοση της. Ακόμα και σήμερα όμως η πλειοψηφία των εφαρμογών μηχανικής μάθησης χρησιμοποιεί μία μόνο υπολογιστική συσκευή (όπως τον επεξεργαστή ή κάποιον επιταχυντή), για να κάνει προβλέψεις, αφήνοντας τις υπόλοιπες υπολογιστικές συσκευές αδρανείς και ανεκμετάλλευτες. Σε αυτή τη δουλειά, προτείνουμε μια διαφορετική προσέγγιση στην οργάνωση και στην ανάθεση των προβλέψεων μοντέλων μηχανικής μάθησης σε ετερογενείς συσκευές. Ο αλγόριθμος που υλοποιεί την ανάθεση των εργασιών στις κατάλληλες συσκευές είναι ικανός να ανταποκριθεί γρήγορα στις δυναμικές διακυμάνσεις πραγματικού χρόνου όπως για παράδειγμα, αυξομειώσεις στην είσοδο του συστήματος, υπερφόρτωση εφαρμογών, και αλλαγές στο υπολογιστικό σύστημα. Τα αποτελέσματα της έρευνας μας δείχνουν οτι ο αλγόριθμος μας είναι ικανός να φτάσει τα μέγιστα ποσοστά απόδοσης ανάμεσα σε διαφορετικά μοντέλα μηχανικής μάθησης, προβλέποντας σωστά την κατάλληλη συσκευή με ποσοστό 92.5%, καταναλώνοντας εώς και 10% λιγότερη ενέργεια.

## Ευχαριστίες

- Ευχαριστώ τον supervisor μου, καθηγητή Ευάγγελο Μαρκάτο για την καθοδήγηση του.

- Ευχαριστώ τον advisor μου, Δρ. Σωτήρη Ιωαννίδη για τις ευκαιρίες που μου έδωσε να δουλέψω σε ενεργά project στο εργαστήριο Κατανεμημένων Υπολογιστικών Συστημάτων. Οι συμβουλές του και η υποστήριξη του ήταν καθοριστήκες για την τεχνική και ακαδημαϊκή μου ανάπτυξη.

- Ευχαριστώ τον ερευνητή και μέντορα, Δρ. Γιώργο Βασιλειάδη για τις ατελείωτες ώρες που σπατάλησε να μου λύνει απορρίες και την πολύτιμη βοήθεια του απο την αρχή εως και σήμερα.

- Ευχαριστώ ολους τους συνάδελφους και φίλους Ντεγιάννη Δημήτρη, Ευα Παπαδογιαννάκή, Γιώργο Χρήστου, Ηλία Παπαδόπουλο, Κώστα Κλεφτογιώργο, Ειρήνη Δέγκλερη, Γιώργο Τσιραντωνάκη, Μιχάλη Διαμαντάρη, Κώστα Σολομό, Αλέξανδρο Κορνιλάκη, Χρήστο Παπαχρήστο, Αντώνη Κριθινάκη για τις συμβουλές, τις συζητήσεις και την στήριξη τους.

- Ευχαριστώ ολους τους ανθρώπους μου που στάθηκαν δίπλα μου τα τελευταία χρόνια, Μιχάλη, Μάνο, Κώστα, Σάββα, Γιώργο και μοιραστήκαμε ολες τις δύσκολες αλλά και τις χαρούμενες στιγμές.

- Ευχαριστώ τους γονείς μου και τα αδέρφια μου που με πίσεψαν και με στήριξαν σε αυτό το εγχείρημα.

- Τέλος ένα μεγάλο ευχαριστώ στην Κωνσταντίνα για την ατελείωτη στήριξη και υπομονή της.

*Στους γονείς μου, Αλέξανδρο και Ελένη*

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

The number of applications that are utilizing machine learning (ML) and deep learning (DL) operations is constantly increasing. A plethora of diversified applications — from autonomous driving to surveillance systems and user profiling — are using machine learning, while the machine and deep learning models themselves are becoming more complex and require significantly more compute power. All these constraints are becoming even more challenging in cases where the applications operate at real-time and require latency-critical requirements.

To cope with these hard performance requirements, it is of paramount importantance to offer methodologies and techniques to process data (either for learning or for inference) at high throughput and/or low latency. Many approaches have been proposed for using specialized accelerators, such as GPUs, to successfully speed up the processing [31]. As a result, there is a shift for many frameworks and libraries to offload the learning and/or inference tasks to high-end GPUs, rather than using the CPU, since the former have more powerful compute resources. Popular libraries and frameworks, such as TensorFlow, Torch, Caffee, MXNet, provide GPU support, as a means to optimize performance, either when performing ML tasks on a batch mode or in a streaming fashion. Other works have also focused on optimizing the key challenges regarding faster inference over streaming data, either by performing computations more efficiently [24], either by performing efficient data movements or transfers, especially when using external accelerators, such as GPUs[19].

The majority of the aforementioned systems focus on the ease of the user as well as on the fastest training of these models, targeting only a single computational device, such as the multicore CPU or a powerful GPU, leaving other devices idle. However, offloading any task to the GPU comes with overheads due to the extra memory copies and trasfers which may result in increases in end-to-end latency or decreased throughput. This includes the extra copies to page-locked memory buffers and the subsequent data transfers to the memory space of the GPU, typically over the PCIe bus. It could be the case though that the computational benefits of the GPU do not pay off these extra data movements; in such cases it

could be more efficient if the computations were performed by the CPU.

Developing an application that can utilize *each* and *every* device effectively and consistently, across a wide range of diverse applications, is highly challenging. Heterogeneous, multi-device systems typically offer system designers different optimization opportunities that offer inherent trade-offs between energy consumption and various performance metrics - in our case, forwarding rate and latency. The challenge to fully tap a heterogeneous system, is to effectively map computations to processing devices, and do so as automated as possible.

In this work we first characterize the performance of a diversified set of machine learning models and we show that some processing devices perform better under different performance metrics (e.g., throughput, latency, and power consumption), while at the same time, these metrics may also deviate significantly among different applications. With these observations in mind, we propose an online, adaptive, scheduler which is able to successfully adjust to different conditions, by taking into account the characteristics and the state of the computational devices, in order to maximize the performance or minimize the latency or energy consumption of our system. Our scheduler can respond quickly to dynamic performance fluctuations that occur at real-time, such as data bursts, application overloads and system changes.

The contributions of our work are:

- We develop a system that runs typical machine learning and deep neural network classification operations on heterogeneous and asymmetric devices, using the OpenCL framework. We further characterize their performance and power consumption, showing that the performance ranking of different computational devices (such as CPUs, high-end GPUs, and integrated GPUs) on different applications is highly varied.

- We propose a machine learning scheduler to mitigate the problem of finding the appropriate device for the classification, given a model architecture and a policy. Our evaluation results show that it is able to select the appropriate device correctly with an accuracy of 92.5% for models that has been trained on, and with an accuracy of 91% for models never seen before.

# Chapter 2

# Background

## 2.1 Background

Typical commodity hardware architectures offer heterogeneity at three levels: *(i)* at the traditional x86 CPU architecture, *(ii)* at an integrated GPU, packed on the same processor die, and *(iii)* at a discrete high-end GPU. All three devices have unique performance and energy characteristics. Overall, the CPU cores are good at handling branch-intensive processing workloads, while discrete GPUs tend to operate efficiently in data-parallel workloads. Between those two, the integrated GPU features high energy efficiency without significantly compromising the processing rate or latency. Typically, the discrete GPU and the CPU communicate over the PCIe bus and they do not share the same physical address space (although this might change in the near future). The integrated GPU on the other hand, shares the LLC cache and the memory controller of the CPU.

### 2.1.1 Architectural Comparison

In Figure 2.1(b), we illustrate the processing scheme that has been used by approaches that utilize a discrete GPU [39, 41, 25, 40].

The majority of these approaches perform a total of seven steps the DMA transaction between the network interface and the main memory, the transfer of the samples to the I/O region, which corresponds to the discrete GPU (this operation traditionally invokes CPU caches, but the cache pollution can be minimized by using *non-temporal* data move instructions) the DMA transaction towards the memory space of the GPU, the actual computational GPU kernel itself and the transfer of the results back to the host memory. All data transfers must operate on fairly large chunks of data, due to the PCIe interconnect inability to handle small data transfers efficiently. The equivalent architecture, using an integrated GPU that is packed on the CPU die, is illustrated on the left side of Figure 2.1. The advantage of this approach is that the integrated GPU and CPU share the same physical memory address space, which allows in-place data processing. This results to fewer data transfers and hence lower processing latency.

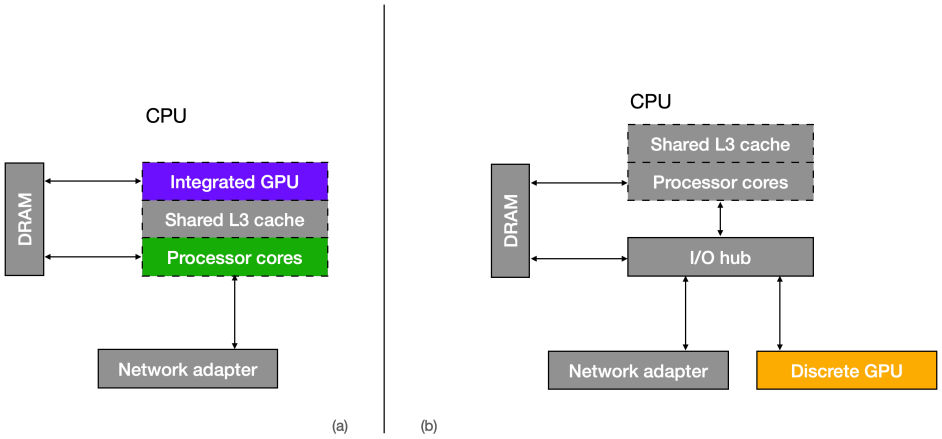



Figure 2.1: Architectural comparison of processing on an (a) integrated and (b) discrete GPU.

### 2.1.2 Quantitative Comparison

An integrated GPU (such as the HD Graphics 4000 we use in this work) has higher energy efficiency as a computational device, compared to modern processors and GPUs. The reason is threefold. First, integrated GPUs are typically implemented with low-power, 3D transistor manufacturing process. Second, they have a simple internal architecture and no dedicated main memory. Third, they match the computational requirements of applications, in which the main bottleneck is the I/O interface and thus, a discrete GPU would be under-utilized. In § 4.2.1 we show, in more detail, the energy efficiency of these devices when executing typical machine learning models.

## 2.2 OpenCL Framework

OpenCL (Open Computing Language) is an open standard for cross-platform, parallel programming of diverse accelerators found in any heterogeneous system such as supercomputers, cloud servers, personal computers, mobile devices and embedded platforms. The term heterogeneous system refers to systems that have numerous devices each with a different underlaying architecture, such as CPUs, GPGPUs, FPGAs and other type of processors and hardware accelerators. OpenCL greatly improves the speed and responsiveness of a wide spectrum of applications in numerous market categories including professional creative tools, scientific and medical software, vision processing, and neural network training and inferencing. [6]. OpenCL's main advantages include (i)*portability*, (ii)*parallel programming* and (iii)*standardized vector processing*.

**Portability:** OpenCL was created from an alliance of the companies: Apple, AMD, IBM, Intel, NVIDIA and Qualcomm, with many more companies adopting the framework. The purpose was to develop a system that would give a huge

flexibility on the users, where one code would run on *all* devices of these vendors. Every vendor that provides OpenCL-compliant hardware also provides the tools that compile OpenCL code to run on the hardware. This means that you can write a code once and it will be translated to the corresponding machine code for every device, eliminating separate compilers or linkers.

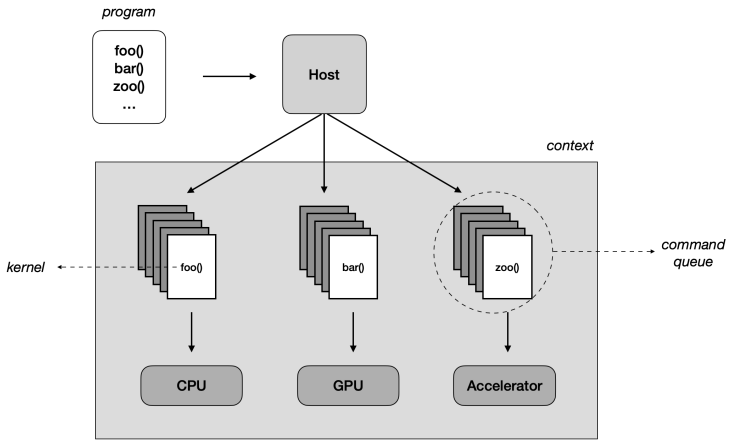



Figure 2.2: OpenCl's architecture

**Parallel Programming:** OpenCL provides *task-parallelism*; many devices can run in parallel the same code, called *kernel*. The full architecture is shown if Figure 2.2. The host creates a *context* where multiple *command-queues* live; each command-queue can contain one or more kernels and is associated with one device of the system. A kernel is created by the host by selecting a function from a *program*, then the kernel is associated with argument data and is being dispatched on a command-queue. Through the mechanism of command queue the host tells to the devices what to do, and when a kernel is enqueued, the device will execute the corresponding function. This level of abstraction enables the execution of kernels in multiple devices in parallel.

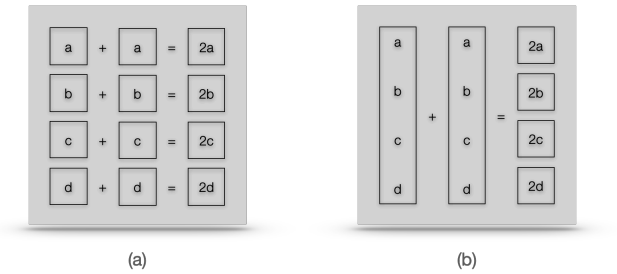



Figure 2.3: Comparison of (a) Scalar operation vs (b)SIMD operation

**Standardized Vector Processing:** The last level of parallelism can be applied inside each kernel. Nearly all modern processors support vector operations and OpenCL offers SIMD programming (Single Instruction Multiple Data). Through data structures that contain multiple elements of the same data type we can execute operations within the same clock cycle. As shown in Figure 2.3 we can add 2 vectors each containing 4 integers in one clock cycle.

## 2.3 Machine Learning

There is a massive variety of Machine Learning models that have been developed throughout the years, each specializing in certain category of problems. Typically Machine Learning consists of two phases, *training* and *inference*. In the training phase we train a model given, usually, large datasets and we try to build a model that can generalize and extract knowledge from this dataset. In this phase the user has to find the appropriate Machine Learning model and has to tune the hyper-parameters of this model such that the model does not overfit or underfit. In the inference phase, we load the trained model on a single device and we make predictions for new unseen data. The training phase is the most time consuming from the user point of view, but the inference phase is the one that going to run and make predictions for much longer time and thus is the most time and resource consuming from the machine point of view. In this work we will focus on the second phase of this procedure, exploring two different type of Machine Learning Neural Networks, *Feed Forward Neural Networks* and *Convolutional Neural Networks*.

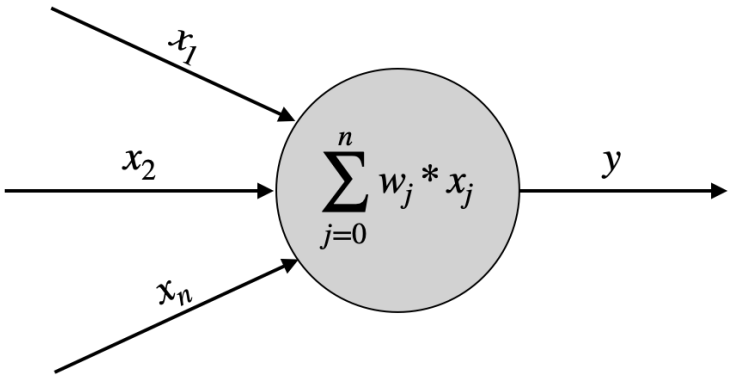



Figure 2.4: Principle element of Neural Network, *Perceptron*

### 2.3.1 Feed Forward Neural Networks

It is the simplest form of Artificial Neural Networks. In Figure 2.4 we can see the *Perceptron*, it takes *inputs* $x_1$, $x_2$,...,$x_n$ and produces a single output y. The computation of the output is an aggregated multiplication of the inputs with real numbers expressing the importance of the input of the respective inputs to the

output, called *weights* $w_1$, $w_2$,...,$w_n$. The neuron's output, can be directly passed on the output as $y = \sum_{j=0}^{n} w_j * x_j$ or it can pass through a nonlinear function such as *relu, tanh* or *sigmoid*. Combining many perceptrons we create a layer of perceptrons and employing many perceptron layers in a sequential way we create multi-layer perceptrons as know as Feed Forward Neural Networks. As we see in Figure 2.5 the Feed Forward Neural Network consists of:

- An input layer
- One or more hidden layers
- An output layer

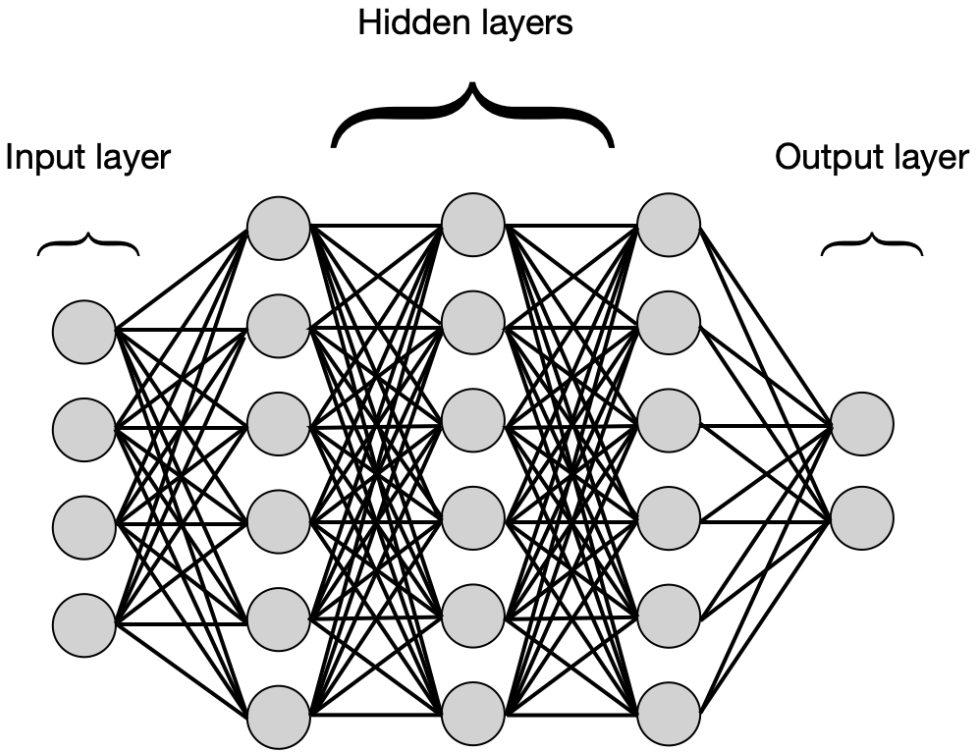



Figure 2.5: Feed Forward Neural Network

### 2.3.2 Convolutional Neural Networks

In deep learning a Convolutional Neural Network is a class of deep neural networks, applied mostly on analyzing visual imagery, financial time series, image and video recognition, speech synthesis and many more. The architecture of a CNN can be seen on Figure 2.6. In reality CNNs consist of two Networks, a Convolution Network and a Feed Forward Neural Network. The advantages of CNNs is two-fold;

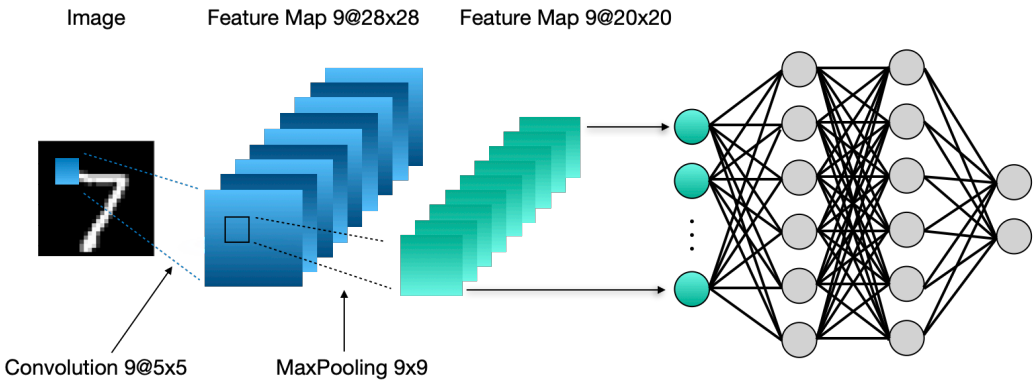



Figure 2.6: Convolutional Neural Network

firstly, the Convolution Network is able to recognise patterns in neighborhoods of 1D or 2D dimensions, thus better recognising patterns in financial series, images, or voice spectrums and secondly, they can scale better than Feed Forward Neural Networks for large images.

For example, images in CIFAR-10 [28], are of size $32 \times 32 \times 3$ (32 wide, 32 high, 3 color channels), so a single fully connected neuron in a first hidden layer of a Feed Forward Neural Network would have 32*32*3 = 3,072 weights. A $200 \times 200$ image, however, would lead to neurons that have 200*200*3 = 120,000 weights, but as we are going to see CNNs down-sampling layer can decrease drastically the first hidden layer of the Feed Forward Neural Network, thus scaling better with the increase of the image size. On top of that Feed Forward Neural Networks do not take into account the spatial structure of data, treating input pixels which are far apart in the same way as pixels that are close together. This ignores locality of reference in image data, both computationally and semantically. Thus, full connectivity of neurons is wasteful for purposes such as image recognition that are dominated by spatially local input patterns. The fundamental block of CNNs is the Convolutional Layer. The layer's parameters consist of a set of *learnable* filters (or kernels) which during the forward pass they convolve across the width and height of the input volume, computing the dot product of the kernel with the input, resulting in a 2-D dimensional activation map of the filter (Figure 2.7), hence the network learns filters that activate when a specific feature is detected. Stacking these activation maps for different filters we end up with the output of the Convolution layer.

The second most important block of CNNs is the Pooling Layer, which essentially is is a non-linear down-sampling procedure. The input of this layer is the activation maps from the convolutional layer, which it partitions in non-overlapping sets of rectangles and for each set computes the maximum. As we have previously mentioned this results in a highly scalable architecture.

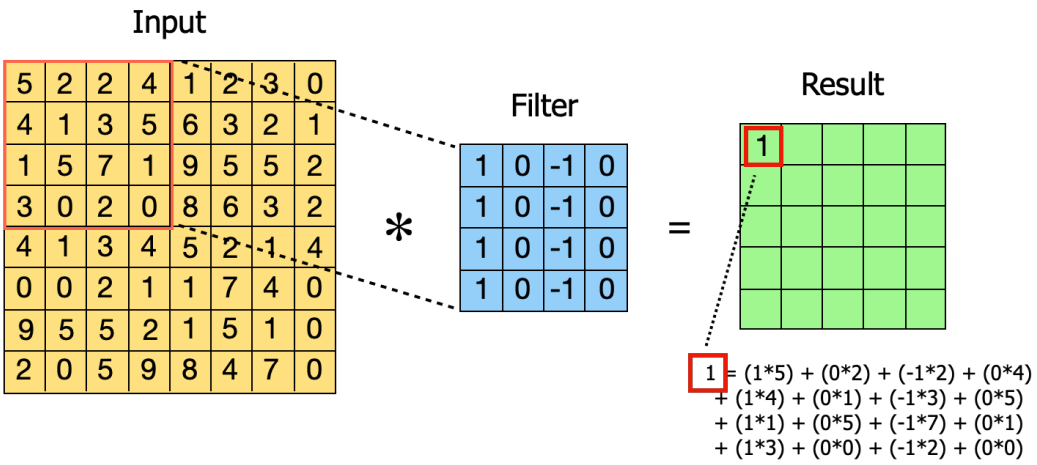



Figure 2.7: Filter & Convolution Operation

### 2.3.3 Metrics and Known Techniques

In this section we are going to introduce some fundamental Machine Learning Metrics and some known techniques vastly applied on the training of ML models, that we are going to use later. In Figure 2.8 we see a confusion matrix of a binary classification model. True Positives are the correct predicted samples of class 0, while True Negatives are the correct predicted samples of class 1. False Positives are the classes predicted as 0 but belong to class 1. Lastly, False Negatives are the samples predicted as 1 but belong to class 0.

1. Precision: It is the fraction of TP divided by (TP + FP). Intuitively it can answer the question of what proportion of the positive identifications was actually correct.
$$\frac{TP}{TP + FP} \tag{2.1}$$

2. Recall: It is the fraction of the TP divided by (TP + FN). Intuitively it can answer the question of what proportion of actual positives was identified correctly.
$$\frac{TP}{TP + FN} \tag{2.2}$$

3. F1-score: It is the harmonic mean of Precision and Recall. Computed as:
$$2 * \frac{Precision * Recall}{Precision + Recall} \tag{2.3}$$

4. Nested cross-validation: Nested cross-validation is an approach to model hyperparameter optimization and model selection that attempts to overcome the problem of overfitting the training dataset.

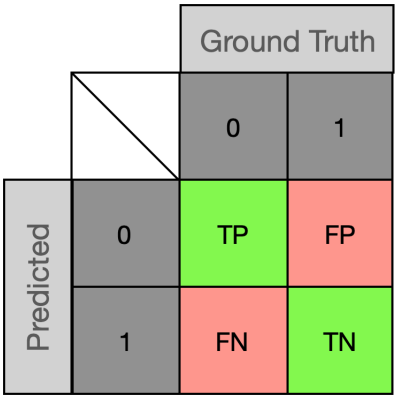



Figure 2.8: Confusion Matrix of Binary Classifier

# Chapter 3

# System Setup

On this chapter we are going to describe the hardware setup that we used for this work. We will also describe the machine learning models that we used and show how we parallelize them using OpenCL, to efficiently execute in each of the three processing devices.

## 3.1 Hardware Setup

Our system contains *three* different, heterogeneous, computational devices: one CPU, one integrated GPU and one discrete GPU. As we can see from Table 3.1 is equipped with one Intel Core i7-8700 Coffee Lake processor, packed with an integrated UHD Graphics 630 GPU and one NVIDIA GeForce GTX 1080 Ti graphics card. The processor contains six CPU cores operating at 3.7GHz, with hyper-threading support, resulting in twelve hardware threads. Overall, The system is equipped with 32GB of dual-channel DDR4-2666 DRAM with 41.6 GB/s throughput. The L3 cache (12MB) and the memory controller are shared across the CPU cores and the integrated GPU. Each CPU core is equipped with 384KB of L1 cache and 1.5MB of L2 cache. The GTX 1080 Ti has 3584 cores in 28 multiprocessors and 11 GB of GDDR5 memory. It is rated at 10609 GFlops, and its Thermal Design Power (TDP) is 250 Watt. The UHD Graphics 630 has 24 execution units, a 64-hardware thread dispatcher and a 100 KB texture cache. The maximum estimated performance of the integrated GPU is rated at 460.8 GFlops on the maximum operating frequency of 1200 Mhz [8]. While Intel does not provide its TDP limit, we estimate that it is close to 20 Watt. For the whole processor die the TDP is 95 Watt.

We notice that our hardware platform exposes an interesting design trade-off: even though the integrated GPU has fewer resources (i.e. hardware threads, execution units, register file) than a high-end discrete graphics card, it is directly connected to the CPU and the main memory via a fast on-chip ring bus, and has much lower power consumption. As we will see in § 4.2.1, this design is well-suited for applications in which the overall performance is limited by the I/O subsystem,

and not by the computational capacity.

Another very interesting observation that we made was that the measurements of the GTX1080Ti were highly affected by the state of the GPU. NVIDIA uses a monitoring system called Boost 3.0 which can under-clock or over-clock the GPU's clocks depending on the workload it has to execute. The GPU's performance is divided in 8 sections ranging from P0 to P7. When the GPU is idle the corresponding performance configuration is set automatically to P7 in order to consume as less energy as possible. As there is no way to control the performance configuration of our GPU and the outcome of our measurements was highly affected we decided to take measurements for both configurations.

| CPU | |
|---|---|
| Model name | Intel Core i7-8700k |
| Clock frequency (GHz) | 3.7 |
| CPUs | 12 |
| Threads per core | 2 |
| Cores per socket | 6 |
| Sockets | 1 |
| **GPU** | |
| Model name | NVIDIA GTX 1080Ti |
| Cores | 3584 |
| Multiprocessors | 28 |
| Base Clock(MHz) | 139 |
| Boost Clock(MHz) | 1582 |
| Memory interface | GDDR5X |
| Memory size(MB) | 11178 |
| FP perf (GFlops) | 10609 |
| TDP (Watts). | 250 |
| **Integrated GPU** | |
| Model name | HD Graphics 630 |
| Clock frequency (MHz) | 1200 |
| Execution Units | 24 |
| TDP(Estimated) | 20 Watt |
| **System Memory** | |
| Size(GB) | 32 |
| Description | DDR4 @ 2666MHz |

Table 3.1: The hardware setup of our base system that we used for our experiments.

## 3.2 Models

For our experiments we use three different models of feed-forward neural networks (FFNN) and two convolutional neural networks (CNN) that cover a small but representative set of machine learning applications.

For the training of these models, we use real datasets (i.e., the Iris dataset [20], Mnist [30], and Cifar-10 [28]), as we describe in more detail below.

**Simple** This model consists of two hidden layers only, each of which contains six nodes in total. It is based on the Iris classification dataset [20] and even though it is one of the simplest feed-forward neural networks it can achieve a great performance with accuracy up to 97%.

**Mnist-Small** The Mnist-Small is a feed-forward neural network that is based on the Mnist dataset, which is a hand written digit database. Overall, it consists of two hidden layers. The first layer consists of 784 nodes, while the second consists of 800 nodes, leading to a 10-node output layer. [37]

**Mnist-Deep** The Mnist-Deep is a feed-forward neural network with six hidden layers, of the following formation: $784 - 2500 - 2000 - 1500 - 1000 - 500$. Similar to Mnist-Small, the output layer consists of 10-nodes. [15]

**Mnist-CNN** The Mnist-CNN is a model that has been trained based on the Mnist dataset. It is a fairly simple CNN model that consists of two VGG blocks, each of which contains one convolution and one pooling layer. The size of the filters of the convolutional layer is $3 \times 3 \times 32$ and the pooling layer size is $2 \times 2$. Finally the FFNN has a Dense layer of 128 nodes, leading to a 10-node output layer. The achieved accuracy of the model was 99.08%.

**Cifar-10** The Cifar-10 is a convolutional neural network that has been trained on the Cifar-10 dataset, which is an image classification database. This model consists of three pairs of Convolutional and Pooling layers (namely VGG blocks), each containing two convolution layers and one pooling layer. The size of the convolutional layers are $3 \times 3 \times 32$ and the pooling layer size is $2 \times 2$. Finally the FFNN has a Dense layer of 128 nodes, leading to a 10-node output layer. The achieved accuracy of the model was 88%.

# Chapter 4

# Implementation

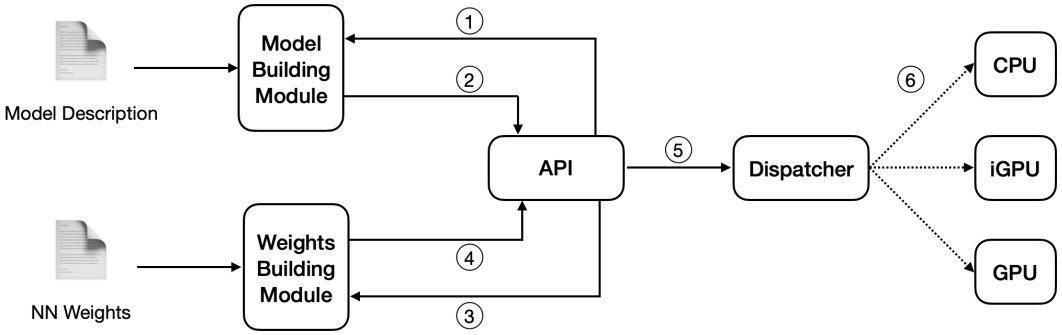
In order to achieve parallel execution and across many devices the classification of these models we used OpenCl. Our aim was to develop a portable implementation of the classification procedure that can run efficiently on heterogeneous devices. We used the OpenCL implementation that comes with NVIDIA CUDA Toolkit 10.0, as well as the Intel OpenCL Runtime for the Core processor family.

## 4.1 Architecture

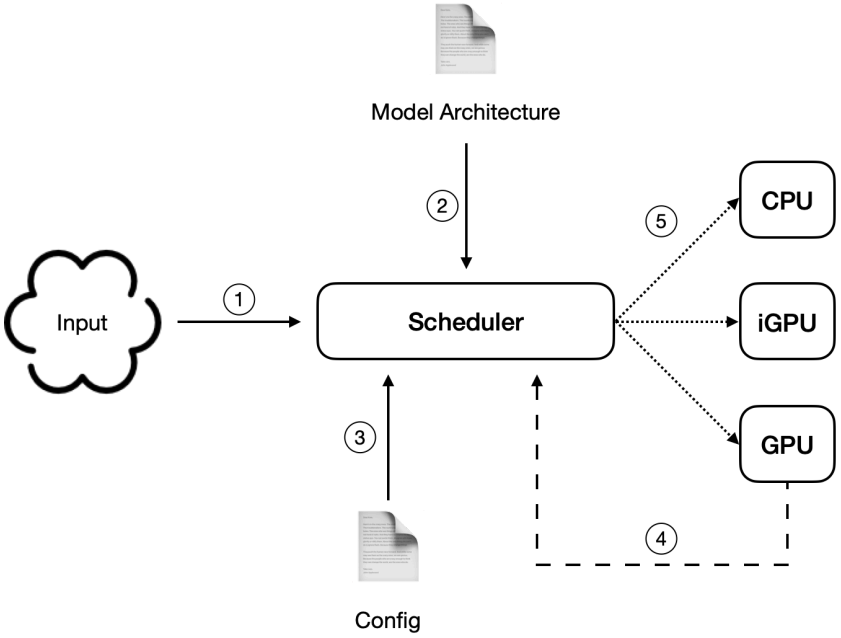
In this section we describe the architecture of our system. Our system consists of *two* phases, Figure 4.1(a) and (b) respectively.

In phase one we create a model given a detailed description. For FFNN we pass the depth of the NN, the number of nodes of each layer and the activation functions. For CNN we give the size and the number of filters of the convolutions, the size of the pooling, the corresponding activation functions and finally a description of a FFNN. All these are handled by the Model Building Module, which builds a model based on these information, this model passed back on the API module. The next step is to load the weights of the NN to the main memory, based on the model that we have build, the Weights Building Module, reads the weights from a file, creates the appropriate buffers and loads the weights in the memory, finally it gives back the buffers into the API Module. The API Module passes the model into the Dispatcher Module, which loads the model and the weights in each device available.

In phase two we ready to run the classification phase in every device available. Firstly, we read from the input (eg. network, file, memory) the data that we want to run the classification for, alongside with the architecture of the specific model and a configuration file specifying if we want achieve a better performance or energy efficiency. The scheduler at this point performs a pci call to check the state of the discrete GPU (idle or not). Based on these information the scheduler determines the correct device and performs the classification.




(a) *Phase one*: In this phase we prepare the system to handle the classification procedure of a model. The interaction of different components is as follows: A model is created from the Model Building Module (1+2). The weights of this model are read from a file and the buffers containing the weights are returned to the central API (3+4). The model and the weights are sent to dispatcher (5) who is responsible to load them on all the available devices (6).




(b) *Phase two*: The system is ready to perform the classification procedure. Input data arrive from various input data streams(1) to the Scheduler alongside with the model architecture (2) and a configuration file which describes the task of the scheduler (Energy efficiency or Best Performance) (3). The scheduler requests from the GPU its current state through a PCI call(4). Based on all these information the scheduler picks the appropriate device to run the classification task (5).

Figure 4.1: Architecture of our system

## 4.2 Parallelization

After the training completes, our system is ready to run the classification phase in every device available. Firstly, we read from the input (e.g., network, file, memory) the data that we want to run the classification for, alongside with the architecture of the specific model and a file containing the policy of our scheduler with three available options, lowest latency, energy efficiency or best throughput. The scheduler at this point performs a PCIe call to check the state of the discrete GPU (idle or not). Based on these information the scheduler determines the correct device and performs the classification.

To execute the Machine Learning applications uniformly across the different devices of our base system, we implement them on top of OpenCL 2.1. Our aim is to develop a portable implementation of each application, that can also run efficiently on each device. Our system runs Linux 5.4.23 with the in-tree i915 driver for the integrated graphics, and nvidia 440.64 driver for the discrete graphics. We use the Intel OpenCL 2.1 SDK for the Core processor family and HD Graphics as well as the OpenCL implementation that comes with NVIDIA CUDA Toolkit 10.0. Due to space constraints we omit the full details of our implementation, and we only list the most important design aspects and optimizations that we have addressed.

We have developed two different compute kernels, one for each type of Neural Networks. In OpenCL, an instance of a compute kernel is called a *work-item*; multiple work-items are grouped together an form *work-groups*. We follow a *thread-per-node* approach, and assign each work-item to handle (at least) one Neural Network layer; More specifically, for the feed-forward neural networks, the nodes of a single layer are computed in parallel, by assigning a separate thread per node. Besides that, we further spawn a second level of parallelization at the sample level, by classifying each sample in parallel. For Convolutional Neural Networks we follow a similar approach, in the convolutional layer we compute in parallel all the convolution operations of a single filter, as well as all the filters of each layer; on the pooling layer all the pooling operations are done in parallel and finally the Fully Connected Neural Network part is computed as described above. With our approach we can classify in parallel up to 256K samples even for computational intensive architectures like Cifar-10 model.

The number of work-items per work-group affects significantly the performance of each device. For example for the GPUs we want many and small work-groups while for CPUs we want less and bigger work-groups. The reason for that is that GPUs have a very fast thread scheduler that can hide the memory-latency of a computation by scheduling another work-group whom data needed for the computation are now ready. On the other hand CPU due to their low latency and their slower thread scheduler can utilise better their resources when their work-groups are as big as possible. From our experiments we have found out that the best configuration for the CPU was 4096 work-items per work-group, whilst the best configuration for the GPU was 256 work-items per work-group, which at the

same time was maximising the available registers per work-item.

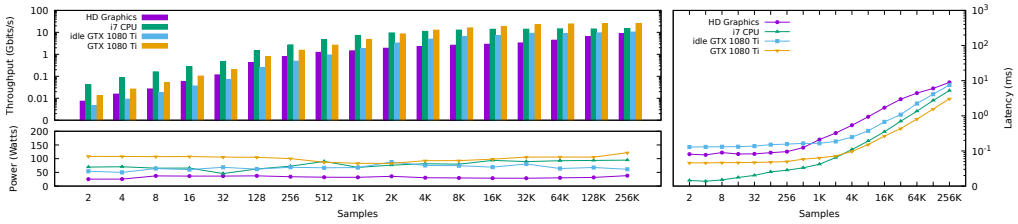
Our devices enable heterogeneity on their memory model as well. For example the global memory of the GPU is physically independent of the memory of the host, requiring a transfer of data from the host memory to the device's global memory through a pci transfer, for every computation. In order to avoid page swapping during the transfers we copy the data that we want to classify on a page-locked buffer. On the other hand CPU's and iGPU's global memory is physically the same memory as the host memory, and we can directly map the corresponding memory buffers using clEnqueueMapBuffer() function to avoid extra copies.

After the data are placed on the global memory of each device, there is another critical aspect that affects the performance of our applications; that is the way the input data are loaded from the global memory of the device. CPUs require row-major order to preserve the cache locality within each thread, while GPUs require column-major order to enable memory loads to be more effective, the so-called memory coalescing. Initially we tried transposing the data in the GPU memory to benefit from the column-major order placement but as we found out the costs of the corresponding data movements, pays off only when accessing the memory with small vector types (i.e. char4); when using the int4 or float4 type though, the overhead is not amortized by the resulting memory coalescing gains, in none of our representative applications. Besides the GPU gains, the CPU enables the use of SIMD units when using the int4 or float4 types, because the vectorized code is translated to SIMD instructions. With all these in mind, we re-design the input process and access the samples using int4 or float4 vector types in a row-major order, for both the CPU and the GPU.

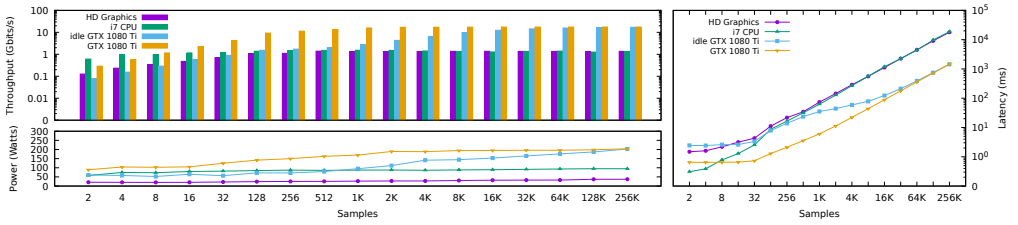
### 4.2.1 Performance Characterization

In this section, we present the performance achieved by our applications. Specifically, we measure the sustained *throughput*, *latency* and *power consumption* for each of the devices available in our base system. To accurately measure the power spent for each device to process the corresponding batch, we measure the power consumption of all the components that are required for the execution. For instance, when we use the GPU for samples inference, the CPU has to collect the necessary data, transfer them to the device (via DMA), spawn a GPU kernel execution, and transfer the results back to the main memory. Instead, when we use the CPU (or the integrated GPU), we exclude the discrete GPU, as it is not needed. By measuring the power consumption of the right components each time, we can accurately and fairly compare different devices.

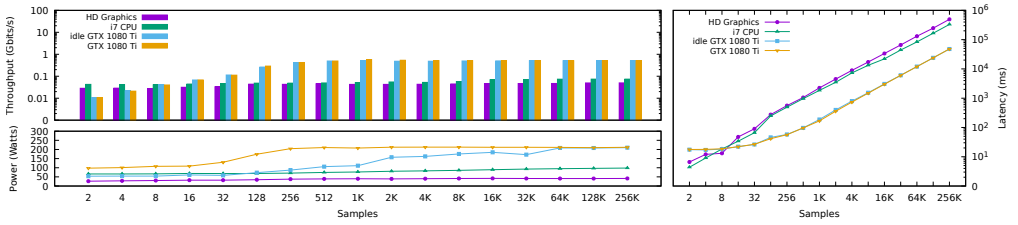
Figures 4.2 and 4.3 summarize the results of our experiments for all the five models that we describe in Section 3.2. In Figure 4.2, we can see the achieved throughput, latency and power for different sample sizes. On the left-hand side we see the achieved throughput and power consumption for the different sample sizes, while on the right-hand side of each subfigure we see the latency for the same



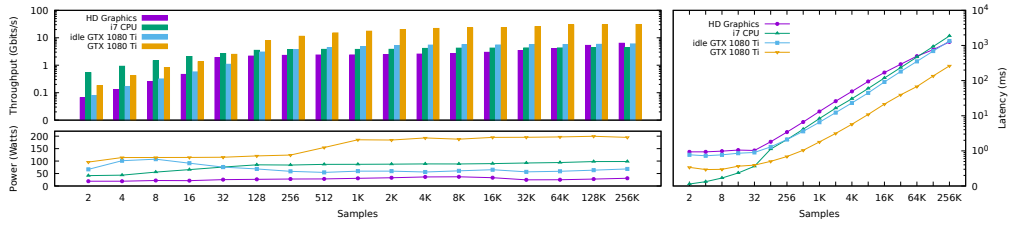

(a) Simple



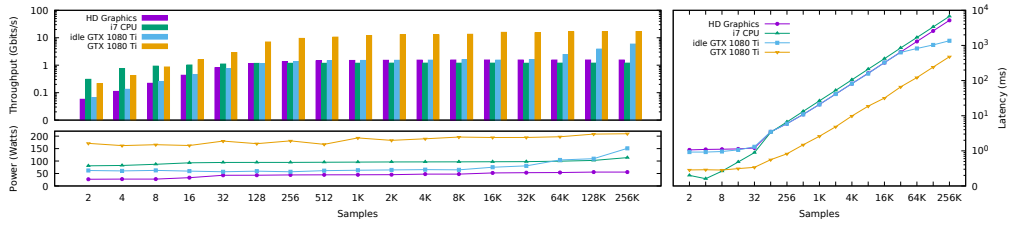

(b) Mnist Small




(c) Mnist Deep

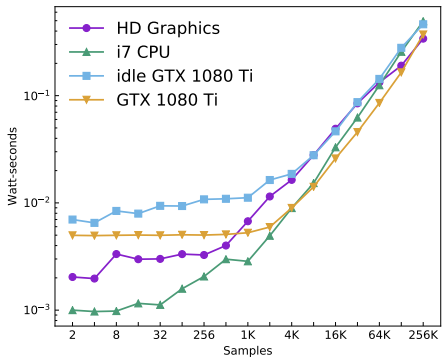



(d) Mnist CNN

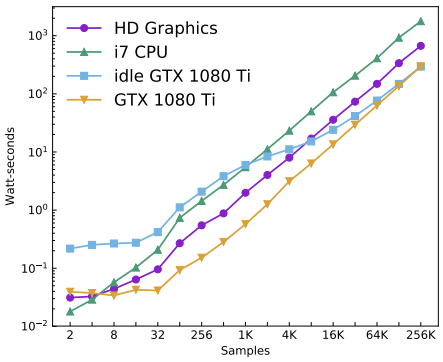



(e) Cifar-10

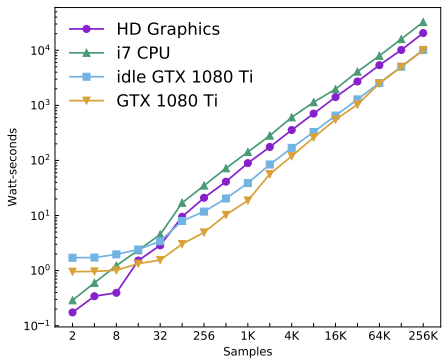
Figure 4.2: Throughput, latency and power consumption for each of the models presented in Section 3.2



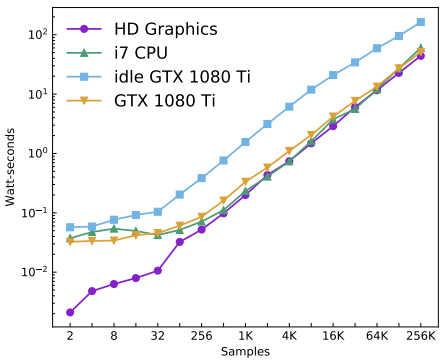

(a) Simple



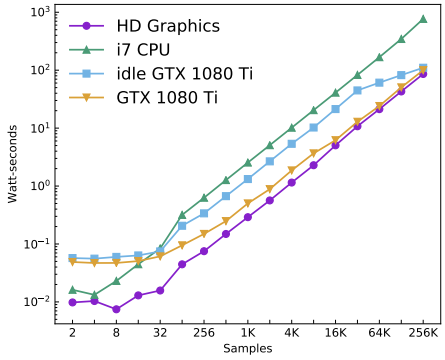

(b) Mnist Small




(c) Mnist Deep




(d) Mnist CNN




(e) Cifar-10

Figure 4.3: Watt-second (Joule) for each of the models presented in Section 3.2

sample sizes. [1]

We observe that the performance of all applications becomes better when the samples size increase. However, the maximum achieved throughput is different for each device, as well as the size of samples that is required to reach it. We can also observe that there is a big variance in the maximum sustained throughput for the different models, ranging from 800 Mbits/s up to 20 Gbits/s for the GPU, and from 50 Mbits/s up to 15 Gbits/s for the CPU. The state of the GPU affects dramatically the sustained throughput in many of the applications, with differences up to 7x. From all the above observations it is obvious that no device performs best across all parameters, in terms of througput. Instead, it is highly affected by the samples size, as well as the computational characteristics of the application, which in our case is the structure of the underlying architecture of the corresponding machine learning model. For example, in Figure 4.2(a) we can see that the CPU performs better only for sample sizes up to 2048 (when the GPU is warmed up);when the GPU starts from an idle state though, the CPU outperforms the GPU for all sample sizes tested. In Figure 4.2(e) we observe that the CPU is better that the GPU for sample sizes up to 8(when the GPU is warmed up); when the GPU starts from and idle state though the CPU is better for sample sizes up to 128. Although there are cases where the state of the GPU is not afecting the outcome, for example in Figure 4.2(c) the CPU is better than GPU for sample sizes up to 8, regardless of the starting state of the GPU.

The sustained latency shows similar tendencies as the throughput. For instance, latency variance is huge throughout the different models, ranging from 1 millisecond up to 16 minutes. Another similarity is that the GPU is suitable for big sample sizes, while the CPU is more suitable for small sample sizes. We can also see that for all the applications from a specific sample size and above there is a linear growth of the latency, this implies that the maximum throughput of this specific device has been achieved. An exception can be observed in Figure 4.2(b) where the idle GPU for sample sizes greater than 512 follows better than linear growth until it matches the warmed-up GPU. For this specific application for sample sizes 64K and above the performance state of the GPU at the start of the classification does not affect the achieved latency. However we can see for smaller sample sizes the best achieved latency is affected by the state of it, specifically if the GPU is not warmed-up the CPU achieves the best performance for sample sizes up to 32, on the contrary CPU is the more suitable device for sample sizes up to 4. In Figure 4.2(d) we can see that the best device considering the GPU starts

[1]In our measurements we have two measurements for the GTX 1080 Ti that are labeled as **idle GTX 1080 Ti** and **GTX 1080**. NVIDIA uses Boost 3.0 tool to automatically lower the GPU clocks when the GPU is idle for better power consumption and it automatically raises the clocks when a task is given to the GPU. In our measurements we have seen that the state of the GPU at the begging of the measurement highly affects the achieved performance of the GPU and in a real case scenario the GPU can reside on either of those states, hence we have two measurements for the GPU, one when the GPU starts from an idle state and one when the GPU is warmed up.

from an idle state is the CPU for sample sizes up to 256, while for GPU starting from P0 performance state the CPU is better for sample sizes up to 32.

In Figure 4.3, we see the Joules that each device needs in order to perform the classification procedure for all the different applications and different sample sizes. Due the heterogeneity of our system we can see that different devices perform better in different applications and configurations; there is no device to rule them all.

Even though the iGPU is the most power efficient device for all applications as shown in Figure 4.2, we get a different impression when we account the Joules consumed per device, depicted in Figure 4.3. The variance of the results is again very big, ranging from 1 mJoules up to 10 KJoules. A general observation is that when the GPU starts from an idle state, it always consumes more energy in all the applications than if it was warmed-up. We can also observe that the CPU is in many applications the worst performing device. An increase in the sample size results to an increase in the consumed Joules, as someone would expect, although we can observe that for each application from a sample size and above there is a linear increase in the consumption, which indicates that the device has reached its maximum computational capacity, however each device reaches that point in a different sample size for each different application. For example, in Figure 4.3(b) the CPU reaches that point in sample size 1024, while the iGPU reached that point in sample size 512. As with the other metrics that we have seen the most appropriate device for a classification is changing based on the sample size in almost all the applications. For example, in Figure 4.3(c) for sample size up to 8 the most appropriate device would be the integrated GPU, but for sample size of 16 and above the most appropriate device would be the GPU. We can see another example in Figure 4.3(b) where the state of the GPU affects a lot the appropriate device for the task, specifically, for sample size 8 up to 4K the iGPU is the most energy efficient device, if the GPU in not warmed-up, while the GPU is the most energy efficient device if is warmed-up.

# Chapter 5

# Efficiency via Scheduling

As we discussed in section § 4.2.1 the performance characterisation indicates that there is not a clear ranking between the benchmarked computational devices. As a consequence of their architectural characteristics, some devices perform better under different metrics, while these metrics may also deviate significantly among different applications. As an example, the i7 CPU performs achieves the best performance on the Iris classification problem, but not on Mnist Deep classification problem.

It is obvious that our system faves heterogeneity in two different levels: *(i)*the different processors and *(ii)*the diverse applications. With these observations in mind, we propose a Machine Learning Scheduler which is able to successfully adapt to different neural network architectures taking into account the state of our computational devices, in order to maximize the performance or minimize the energy consumption of our system. In the next sections we are going to describe how we accomplished that.

Table 5.1: Different Hyperparameters of our Random Forest Classifier

| **Hyperparameters** | **Description** | **Values** |
|---|---|---|
| `n_estimators` | Number of trees in the forest | {5,10,15,20,25,30,35 40,45,50,100,200} |
| `max_depth` | Maximum depth of the tree | {3,4,5,6,7,8,9,10} |
| `criterion` | Function to measure the quality of a split | {"entropy", "gini"} |
| `min_samples_leaf` | Minimum number of samples required to be at a leaf node | {1,2,3,4,5,10,15} |

## 5.1 Online Scheduler

Our proposed scheduler, shown in Figure 4.1(b), is based on machine learning algorithms to make decisions. The motivation to use machine learning is the fact that the neural network models that may need to execute, usually, have a strong diversity. Additionally, it is also typical to dynamically add models for which we do not have any measurements; a static approach does not scale easy, in contrast with our proposed system that is able to learn and extract knowledge from a dataset. Our aim is to train a model that would be able to learn and predict the appropriate device on which a classification model will run. We also want to have control regarding the target that we want to achieve, i.e., best throughput, best latency or best energy efficiency.

In order to find a suitable machine learning model for our scheduler, we tried different approaches, including Linear Regression, SVM, k-Nearest Neighbors, Random Forest and Feed Forward Neural Network. After careful evaluation, we found that the Random Forest classifier performs better in terms of accuracy and performance; in Section 6, we present the evaluation results of the different models and corresponding tradeoffs.

### 5.1.1 Data Augmentation & Preparation

One important design decision is the representation of the data that will be used for the training of the scheduler. As we discuss in Section 4.2.1, the most important parameters is the samples size and the state of the GPU; both parameters affect significantly the selection of the appropriate device, as such they are mainly used for the training. To remedy the limited dataset that we have (i.e., in terms of quantity we had only 340 samples) as well as the lack of variety in Machine Learning models (i.e., we had only the 5 models of section 3.2) we measure 16 more models with different architectures. With each of these models we tried to capture how the different parameters of FFNN and CNNs affect the sustained metrics. For example, a FFNN has two parameters that affect the performance of each device: (i) the depth of the model and (ii) the size of the layers. CNN has four parameters that affect the performance of the devices: (1) The number of consecutive VGG blocks, (2) the size of convolutions, (3) the size of pooling, and (4) the number of convolutions layers per VGG block. With the models that we used to augment our data, we capture all the different parameters of these architectures. Overall, we end up with 1480 samples which we use to train our scheduler. The corresponding classes for *CPU*, *GPU*, and *iGPU* ended in an imbalanced state, with 30% from first class, 40% from the second class and 30% from the third class.

For the representation of the feed-forward neural networks, we use two parameters, one representing the network depth and another representing the total number of neurons. Lastly, for the representation of the convolutional neural networks, we have four additional parameters that represent the number of the VGG blocks, the convolutions per VGG block, the size of the convolution filter and the

size of the pooling layer.

| Model | Accuracy | Training Time | Classification Time |
|---|---|---|---|
| Model | Accuracy | Training Time | Classification Time |
| Baseline (Random Selection) | 41% | n/a | 0ms |
| Linear Regression | 77.94% | 15 sec | **0.7ms** |
| SVM | 53.38% | 49 min 7sec | 0.77ms |
| k-NN | 62.64% | 5 sec | 1.3ms |
| Feed Forward Neural Network | 52.62% | 10 sec | 0.77ms |
| Random Forest | **93.22%** | 26 sec | 3.35ms |
| Decision Tree | 92.01% | **0.5 sec** | 0.9ms |

Table 5.2: All Machine Learning Models that we tried

### 5.1.2 Train the Scheduler

It is well known that the Random Forests do not perform pretty well on imbalanced data, our solution for this issue is two-fold. Firstly, we will **not** report the Accuracy of the System for a proof that our scheduler works well, rather we are going to report F1-score, Precision and Recall. Secondly we did a Stratified k-Fold splitting in our dataset to ensure that the classifier was trained with balanced data. More specifically, to train our Random Forest classifier we did a Stratified k-Fold Nested cross validation. Stratified k-Fold was applied because the classes of our dataset were imbalanced; we did cross-validation to overcome the known overestimating problem of classifiers, and we did it in a nested way to find the best hyperparameters of the classifier.

For the training of all the ML models we used the Scikit-learn programming framework on top of Python 3.6. As we described in § 2 the nested cross validation is an iterative process, but we can parallelize the execution of each of the outer folds as well as we can parallelize the inner folds as well. In our base system each outer fold takes around 20 seconds to train, but due to the fact that all the jobs are running in parallel the total training time of the Random Forest is 26 seconds. The hyperparameters that we tried are shown on Table 5.1.

# Chapter 6

# Evaluation

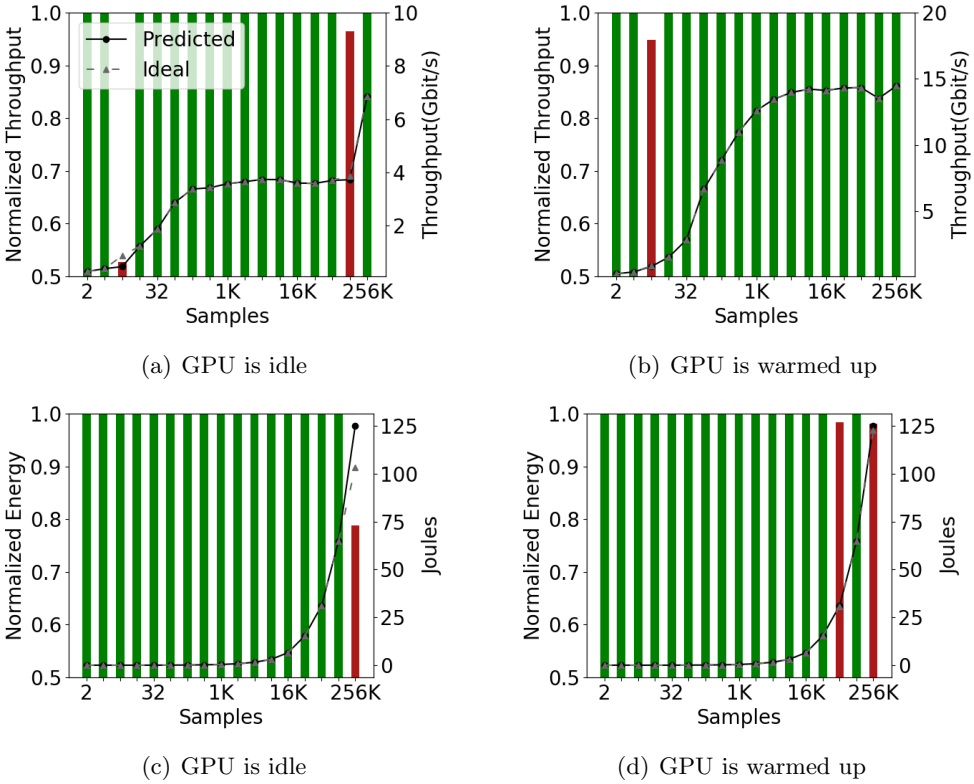
In this section we evaluate our scheduler in terms of performance and accuracy, using different machine learning models, presented in Table 5.1.1. As we can see, these models have different benefits and tradeoffs that need to be carefully considered according to the application requirements. For instance, Linear Regression provides the fastest inference execution times, while also requiring very small time for the training. However, its accuracy score is not high enough to be used as our main classifier.

Decision Trees, on the other hand, provide the fastest training time and one of the fastest inference times too, while maintaining an accuracy of 92.01%. However, further experiments showed that this algorithm performs poorly on totally unseen machine learning models (i.e. accuracy on *unseen* data is 70.2%). Random Forest, on the other hand, achieves the best accuracy compared to the other models, even though this comes at a cost of extra time needed to perform the classification. To get more clear insights on its accuracy though, we further evaluate its F1-score. As we have already mentioned in Section 2, the F1-score correlates both precision and recall metrics, which gives a better understanding of how a model performs. As we can see in Table 6.1, Random Forest performs really well for scheduling decision, both in terms of precision and recall.

Moreover, Random Forest is very efficient when making predictions for machine learning models that *are not* in the training dataset. Figure 6.1 plots the corresponding predictions for matching maximum performance and best energy efficiency, as well as the affected performance loss as a result of the wrong predictions. The green bars indicate that the scheduler made a correct prediction and the red ones indicate the wrong predictions. For example, in Figure 6.1(a) we can see that the scheduler made a wrong prediction for sample size 8 and the achieved

Table 6.1: Scheduler efficiency when using Random Forest classifier

| F1-score | Precision | Recall |
|---|---|---|
| 93.51% | 93.22% | 93.21% |




(a) GPU is idle

(b) GPU is warmed up

(c) GPU is idle

(d) GPU is warmed up

Figure 6.1: Throughput achieved and energy consumed with the predictions of our scheduler

throughput is 43% lower than the ideal throughput, while for sample size 128K the achieved throughput is only 4% lower. Our scheduler achieved a combined score of 91% for the two different policies, while the performance loss due to wrong predictions is less than 5%. As such, we can conclude that it can achieve highly accurate predictions, even for cases that has not seen before.

# Chapter 7

# Related Work

A scheduler is one of the fundamental elements of computer science, and they can be found in many different layers of the software stack, from an operating system [32] up to clusters scheduling [17], [18]

**Acceleration of ML Applications:** Many works focus on accelerating the training and inference of ML Applications, one approach is the sparcification of DNNs [22], [33], [36] where by decreasing the weights of DNNs up to 90% the inference performance can be improved significantly. Some other approaches focus on decreasing inference latency by changing the DNN itself. For instance, binarized neural networks perform compression and pruning on the models reduce memory consumption and computations for each inference [16]. Eyeriss proposes a dataflow that exploits local data reuse and minimizes data movements in the neural network [13]. All these approaches are orthogonal to our work and can be adapted as device-specific optimizations.

Another popular approach is the development of specialized hardware such as NVIDIAs Tensor Cores [5], Google's Tensor Processing Unit [2], Microsoft's specialised hardware [4], Tesla's autonomous system specialized hardware [7] and many more [29], [1], [3] in order to accelerate the training and inference of the DNNs. With all these options available the heterogeneity in the hardware level is increased, thus approaches like ours are even more needed.

**ML Frameworks:** There is a plethora of ML Frameworks that exist today that aim to facilitate the complicated data analysis process and to propose integrated environments on top of standard programming languages. Although they are designed to do the same thing, developing ML applications, their focuses are slightly different. For example, Scikit-Learn [35], Pytorch [34] and Keras [14] are designed keeping in mind the user convenience, making the development of an ML application as easy as possible for the user. Deepleaning4j [38] and SparkML [21] are designed in order to handle enormous datasets and distributed training. Tensorflow [9] is designed for large-scale distributed training and inference. Caffe [26] is designed with expression, speed, and modularity in mind. Our work is orthogonal to all these Frameworks, and can be added as a feature.

**Performance prediction:** Many works focus on trying to predict the performance of a GPU. In [12], the authors try to predict the performance of the GPU, based on the corresponding CPU performance; their system then tries to select the best device on a set of different applications. Other works focus on heterogeneous task partitioning by predicting the performance of the devices, using compiler-based and neural network approaches accordingly [23, 27]. In [10] they try to predict the performance of GEMM for heterogeneous devices using machine learning models such as SVMs and neural networks. Lastly, in [11] they try to predict the performance of heterogeneous systems using machine learning models and the hardware specifications of each device. A major different of our approach with the majority of these works is that they do not support different policies when scheduling the applications for execution. Moreover, their schedulers are not adaptive to changes or fluctuations.

# Chapter 8

# Discussion

In this section we are going to discuss about the limitations of our work, the future work as well as some thoughts we have regarding the Machine Learning Scheduler.

## 8.1 Machine Learning Scheduler

Another approach that we could use to find the appropriate device would be to train a Machine Learning model to predict the achieved Throughput and the consumed Joules for all the different configurations. With this approach we believe that a Neural Network would be able to predict very accurately the correct device for the given task.

## 8.2 Concurrent Workloads

One limitation of our architecture is the lack of optimization capabilities for concurrent running applications. The optimal parallel scheduling of an arbitrary application mixture is a highly challenging problem, mainly due to the unknown interference effects. These effects include but are not limited to: contention for hardware recourses (e.g. shared caches, I/O interconnects, etc.), software resources, and false sharing of cache blocks. We believe that our machine learning scheduler will be able to handle this scenarios. In this work we solely focus on optimizing the performance of a single application that executes on a set of computing devices. As part of our future work we plan to experiment with application multiplexing and investigate the feasibility of a more generic energy-aware scheduler.

# Chapter 9

# Conclusion

In this work we address the problem of improving the efficiency of machine learning classification on commodity, off-the-self, heterogeneous architectures. Heterogeneous systems can provide substantial performance improvements, but only when scheduling appropriately. A static approach can lead to suboptimal performance when the input data rates, system or application changes. To remedy this, we propose an online adaptive scheduling algorithm, that can *(i)* respond effectively to relative performance changes, and *(ii)* significantly improve the energy efficiency of machine learning classification inference workloads. Our system is able to efficiently utilize the computational capacity of its resources on demand, resulting in predicting correctly the appropriate device with an accuracy of 92.5%, while consuming up to 10% less energy.

As part of our future work we plan to extend our architecture to support more convolutional neural networks models, such as dilated convolutions, convolution padding and more. We also plan to extend our architecture to support execution of different machine learning inference operations concurrently.

# Bibliography


[1] Deep learning on mppa. Available: https://indico.cern.ch/event/395242/attachments/791742/1085286/DEEP_LEARNING_ON_MPPA_MANYCORE_PROCESSOR_V2.2.pdf. Accessed on Oct. 2, 2020.

[2] Google tpu system architecture. Available: https://cloud.google.com/tpu/docs/system-architecture. Accessed on Oct. 2, 2020.

[3] Ibm finds killer app for truenorth neuromorphic chip. Available: https://www.top500.org/news/ibm-finds-killer-app-for-truenorth-neuromorphic-chip/. Accessed on Oct. 2, 2020.

[4] Microsoft unveils project brainwave for real-time ai. Available: https://www.microsoft.com/en-us/research/blog/microsoft-unveils-project-brainwave/. Accessed on Oct. 2, 2020.

[5] Nvidia tensor cores. Available: https://www.nvidia.com/en-us/data-center/tensor-cores/. Accessed on Oct. 2, 2020.

[6] OpenCL. Available: http://www.khronos.org/opencl/. Accessed on Dec. 7, 2016.

[7] Tesla vaunts creation of 'the best chip in the world' for self-driving. Available: https://techcrunch.com/2019/04/22/tesla-vaunts-creation-of-the-best-chip-in-the-world-for-self-driving/. Accessed on Oct. 2, 2020.

[8] *Intel HD Graphics DirectX Developer's Guide*, 2010.

[9] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan

Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[10] S. Agrawal, A. Bansal, and S. Rathor. Prediction of sgemm gpu kernel performance using supervised and unsupervised machine learning techniques. In *2018 9th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, 2018.

[11] M. Amaris, R. Y. de Camargo, M. Dyab, A. Goldman, and D. Trystram. A comparison of gpu execution time prediction using machine learning and analytical modeling. In *2016 IEEE 15th International Symposium on Network Computing and Applications (NCA)*, 2016.

[12] I. Baldini, S. J. Fink, and E. Altman. Predicting GPU Performance from CPU Runs Using Machine Learning. In *26th International Symposium on Computer Architecture and High Performance Computing*, 2014.

[13] Y. Chen, J. Emer, and V. Sze. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 367–379, 2016.

[14] Francois Chollet et al. Keras, 2015.

[15] Dan Claudiu Ciresan, Ueli Meier, Luca Maria Gambardella, and Jürgen Schmidhuber. Deep big simple neural nets excel on handwritten digit recognition. *CoRR*, abs/1003.0358, 2010.

[16] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830*, 2016.

[17] Christina Delimitrou and Christos Kozyrakis. Qos-aware scheduling in heterogeneous datacenters with paragon. *ACM Trans. Comput. Syst.*, 31(4), December 2013.

[18] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, page 127–144, New York, NY, USA, 2014. Association for Computing Machinery.

[19] A. Dhakal and K. K. Ramakrishnan. Netml: An nfv platform with efficient support for machine learning applications. In *2019 IEEE Conference on Network Softwarization (NetSoft)*, pages 396–404, 2019.

[20] Dheeru Dua and Casey Graff. UCI machine learning repository, 2017.

[21] Xiangrui Meng et al. Mllib: machine learning in apache spark. *The Journal of Machine Learning Research*, 17(1), 2016.

[22] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. *arXiv preprint arXiv:1803.03635v5*, 2019.

[23] A. E. Helal, W. Feng, C. Jung, and Y. Y. Hanafy. AutoMatch: An automated framework for relative performance estimation and workload distribution on heterogeneous HPC systems. In *2017 IEEE International Symposium on Workload Characterization (IISWC)*, 2017.

[24] N. Ho and W. Wong. Exploiting half precision arithmetic in nvidia gpus. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2017.

[25] Muhammad Jamshed, Jihyung Lee, Sangwoo Moon, Insu Yun, Deokjin Kim, Sungryoul Lee, Yung Yi, and KyoungSoo Park. Kargus: a Highly-scalable Software-based Intrusion Detection System. In *Proceedings of the 19th ACM Conference on Computer and Communications Security*, 2012.

[26] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[27] Klaus Kofler, Ivan Grasso, Biagio Cosenza, and Thomas Fahringer. An automatic input-sensitive approach for heterogeneous task partitioning. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS, 2013.

[28] Alex Krizhevsky. Learning multiple layers of features from tiny images. 2009.

[29] Griffin Lacey, Graham W. Taylor, and Shawki Areibi. Deep learning on fpgas: Past, present, and future. *arXiv preprint arXiv:arXiv:1602.04283*, 2016.

[30] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010.

[31] Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph E Gonzalez, and Ion Stoica. Tune: A research platform for distributed model selection and training. *arXiv preprint arXiv:1807.05118*, 2018.

[32] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. The linux scheduler: A decade of wasted cores. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, New York, NY, USA, 2016. Association for Computing Machinery.

[33] Eran Malach, Gilad Yehudai, Shai Shalev-Shwartz, and Ohad Shamir. Proving the lottery ticket hypothesis: Pruning is all you need. *arXiv preprint arXiv:2002.00585v1*, 2020.

[34] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[35] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[36] Vivek Ramanujan, Mitchell Wortsman, Aniruddha Kembhavi, Ali Farhadi, and Mohammad Rastegari. What's hidden in a randomly weighted neural network? *arXiv preprint arXiv:1911.13299v2*, 2020.

[37] Patrice Y. Simard, Dave Steinkraus, and John Platt. Best practices for convolutional neural networks applied to visual document analysis. Institute of Electrical and Electronics Engineers, Inc., August 2003.

[38] Eclipse Deeplearning4j Development Team. ND4J: Fast, Scientific and Numerical Computing for the JVM. 2016.

[39] Giorgos Vasiliadis, Spiros Antonatos, Michalis Polychronakis, Evangelos P. Markatos, and Sotiris Ioannidis. Gnort: High Performance Network Intrusion Detection Using Graphics Processors. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection*, 2008.

[40] Giorgos Vasiliadis, Lazaros Koromilas, Michalis Polychronakis, and Sotiris Ioannidis. GASPP: A GPU-Accelerated Stateful Packet Processing Framework. In *Proceedings of the 2014 USENIX Annual Technical Conference*, 2014.

[41] Giorgos Vasiliadis, Michalis Polychronakis, and Sotiris Ioannidis. MIDeA: A Multi-Parallel Intrusion Detection Architecture. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, 2011.