The Best of Many Worlds: Scheduling Machine Learning Inference on CPU-GPU Integrated Architectures

Rafail Tsirbas*, Giorgos Vasiliadis*†, Sotiris Ioannidis*‡
*Foundation for Research and Technology - Hellas
{tsirbas, gvasil, sotiris}@ics.forth.gr
†Hellenic Mediterranean University
gvasil@hmu.gr
‡Technical University of Crete
sotiris@ece.tuc.gr

***Abstract*—A plethora of applications are using machine learning, the operations of which are becoming more complex and require additional computing power. At the same time, typical commodity system setups (including desktops, servers, and embedded devices) are now offering different processing devices, the most often of which are multi-core CPUs, integrated GPUs, and discrete GPUs. In this paper, we follow a data-driven approach, where we first show the performance of different processing devices when executing a diversified set of inference engines; some processing devices perform better for different performance metrics (e.g., throughput, latency, and power consumption), while at the same time, these metrics may also deviate significantly among different applications. Based on these findings, we propose an adaptive scheduling approach, tailored for machine learning inference operations, that enables the use of the most efficient processing device available. Our scheduler is device-agnostic and can respond quickly to dynamic 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, by predicting correctly the optimal processing device with an accuracy of 92.5%, with energy savings up to 10%.**

I. 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 content understanding to object detection and speech translations — are using machine learning, while the machine and deep learning models themselves are becoming more complex and require significantly more computing power. Many approaches have been proposed for using specialised accelerators to successfully speed up the processing, using either discrete GPUs [1], [2] or even integrated GPUs [3]. Other works have also focused on optimizing the key challenges regarding faster inference over streaming data, either by performing computations more efficiently [4], or by performing efficient data movements or transfers when using external accelerators, such as GPUs[5]. However, the majority of the aforementioned systems target only the most powerful device, leaving other devices idle and potentially underutilizing the available computational power.

Besides the unique performance characteristics of each processing device, which is actually a direct consequence of their underlying architectural design, several other factors can affect their efficiency and effectiveness. As a matter of fact, dynamic performance fluctuations that occur at run time, such as those caused by changes to a processor's clock frequency or the device state, can significantly affect the resulted performance. Last but not least, data variability in the form of overloads or those caused due to diurnal patterns can have a major consequence in the overall power consumption – e.g., selecting a low-end device in cases where the data load is low would have significantly lower energy requirements.

In this paper, we first characterise the performance of a diversified set of machine learning models; this analysis shows 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 scheduler that is able to successfully adapt to different conditions. Our scheduler takes into account the characteristics and the state of the computational devices, in order to maximise the performance or minimise the latency or energy consumption of the underlying system. In addition, it *can respond quickly to dynamic performance fluctuations* that occur at real-time, such as data bursts, application overloads and system changes. Finally, we note that our scheduler is *device-agnostic*; even though we use a fixed set of processors and co-processors in this paper (which are representative though in a typical heterogeneous production system), our system can similarly operate when any other processors or co-processors are present (i.e., FPGAs, NPUs, or DSPs).

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 processors and co-processors, using the OpenCL framework. We further characterize their performance and power consumption systematically, 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 an adaptive scheduler that is able to mitigate the problem of finding the appropriate device for the classification, given a model architecture and a policy. Our evaluation results show that our scheduler 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.

II. Background

It is currently typical in commodity system setups (both in user desktop/laptop machines and cloud computing providers, even in mobile and embedded devices) to offer different, heterogeneous, processing devices. Such devices usually include the traditional x86 CPU architecture, an integrated GPU (hereafter iGPU) that is packed on the same processor die, and a discrete high-end GPU. These devices have unique performance and energy characteristics and are optimized for different operations. For instance, the CPU cores are good at handling branch-intensive processing workloads, while discrete GPUs tend to operate efficiently in data-parallel workloads. Between these two, the iGPU offers satisfying levels of processing rate and latency, while enabling high energy efficiency. Moreover, the iGPU shares the LLC cache and the memory controller of the CPU, contrary to the discrete GPU that communicates with the CPU over the PCIe bus.

Such heterogeneity in hardware level, enables different opportunities and different challenges in executing different workloads. These challenges can further exacerbate when take into perspective the heterogeneity that may arise at the application level (e.g., in terms of memory- or compute-intensiveness) or at the data that need to be processed (e.g., in terms of volume, velocity, or complexity). In the case of machine learning execution this can be reflected by the number of layers and/or nodes per layer needed, as well as by the complexity and size of the incoming data that need to be processed.

A. Architectural Heterogeneity

The approaches that utilize a discrete GPU typically perform a total of four steps: the copy of data to the I/O region that 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 iGPU that is packed on the CPU die, has the advantage of a unified physical memory between the iGPU and the CPU, which allows in-place data processing. This results to fewer data transfers and and also has lower power consumption compared to the discrete GPU setup due to the better energy effiency of the iGPU and the absence of hardware resources (such as the I/O Hub).

1) Quantitative Comparison: An iGPU (such as the UHD Graphics 630 GPU we use in this work) has higher energy efficiency as a computational device, compared to modern CPUs and discrete GPUs. The reason is threefold. First, iGPUs 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 section IV-C we show, in more detail, the energy efficiency of these devices when executing typical machine learning models.

B. Machine Learning

There are many different varieties of machine learning models that have been developed throughout the years, each of which is specialized in certain categories of problems. Typically, machine learning consists of two phases: (i) the *training* phase, and (ii) the *classification* or *inference* phase. In the training phase, a model is trained using, usually, large datasets, while trying to generalize and extract knowledge out of it. At this phase, it is important to find the appropriate machine learning model and tune its hyperparameters so that the model does not overfit nor underfit. In the inference phase, the trained model is used to make predictions for new, unseen, data. Even though the training phase is typically the most time consuming task, it generally takes place offline; on the other hand, the inference phase is used to run and make real-time predictions for much longer time and thus is the most time and resource consuming from the machine point of view. This paper focuses on the latter phase, exploring two different type of machine learning neural networks, namely the *feed-forward neural networks* and the *convolutional neural networks*. We choose these two types of machine learning models for two reasons. First, they are two of the most computationaly intensive models in the machine learning area. Second, they are vastly used by many applications, mainly due to their ability to model very difficult tasks (e.g., image and speech recognition, speech synthesis and many more).

1) Feed-Forward Neural Networks: A feed-forward neural network (FFNN) is the simplest form of artificial neural network, and consists of *perceptrons*; a perceptron takes as input a set of x_1, x_2, ..., x_n elements 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 respective inputs to the output, called *weights* w_1, w_2,...,w_n. The neuron's output, can be directly passed at the output as $y = \sum_{j=0}^{n} w_j * x_j$ or it can be passed through a nonlinear function such as *relu, tanh* or *sigmoid*. Many of these perceptrons form a layer of perceptrons, while employing

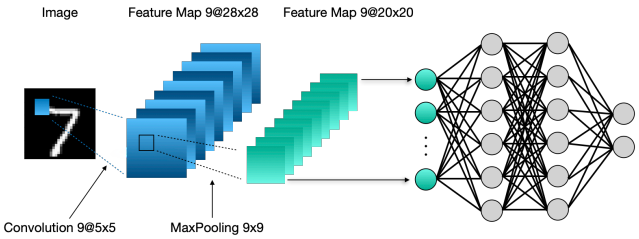


Fig. 1. A typical structure of a convolutional neural network

many of these layers form multilayer perceptrons, known as feed-forward neural networks. A typical feed-forward neural network conists of: (i) an input layer, (ii) one or more hidden layers, and (iii) an output layer.

2) Convolutional Neural Networks: A convolutional neural network (CNN) is a class of deep neural network that is mostly applied, for example, when analyzing visual imagery, financial time series, speech synthesis, image and video recognition. The advantage of CNNs is that they can recognise patterns in neighbors of areas in 1, 2 or 3 dimensions; this feature enables better recognition of patterns in financial series, images, or voice spectrums. The typical structure of a CNN is shown in Figure 1. As we can see, the input is convolved with a matrix, resulting in multiple feature maps, which are then reduced with a max-pooling operation; the final results, called features, are flattened and are used as the input layer of a feed-forward neural network. The pair of a convolution layer with a pooling layer is going to be referred as a Visual Geometry Group (VGG) block.

III. System Setup

We now describe the hardware setup, and our power instrumentation and measurement scheme. Our scheme is capable of accurately measuring the power consumption of various hardware components, such as the CPU and GPU, in real time. We also describe the machine learning models that we use for this work and show how we parallelize them using OpenCL, in order to execute efficiently in different processing devices.

We note that in our system we use a set of different processors and co-processors, which are representative in a typical heterogeneous production system. However, our system can similarly operate when other processors or co-processors are present (i.e., FPGAs, NPUs, or DSPs); the scheduler we propose *is device-agnostic in that sense*.

A. Hardware Platform

Our base system is equipped with one Intel Core i7-8700 Coffee Lake processor and one NVIDIA GeForce GTX 1080 Ti graphics card. The CPU contains six cores operating at 3.7GHz, with hyper-threading support, resulting in twelve hardware threads, and an integrated UHD Graphics 630 GPU. Overall, our system contains *three* different, heterogeneous, computational devices: one CPU, one iGPU and one discrete GPU. 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 iGPU. Each CPU core is equipped with 384KB of L1 cache and 1.5MB of L2 cache. The GTX 1080 Ti has 3584 cores that are organized in 28 multiprocessors and is also equipped with 11 GB of GDDR5 memory. It is rated at 10.6 TFlops and its Thermal Design Power (TDP) is 250 Watt. The UHD Graphics 630 has 24 execution units, a 64-hardware thread dispatcher and 100 KB of texture cache. The maximum estimated performance of the iGPU is rated at 460.8 GFlops on the maximum operating frequency of 1200 MHz [6]. 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 tradeoff: even though the iGPU 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 Section IV-C, 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.

1) Power Instrumentation: To accurately measure the power consumption of our hardware system, we use a software approach. For the GTX 1080Ti, we use `nvidia-smi` which is designed to aid in the management and monitoring of NVIDIA GPU devices. [7] From Kepler architecture and beyond `nvidia-smi` supports the power management utility which is able to report the board's power draw in a live manner. For i7-8700k we use a similar approach; Intel Processor Counter Monitor (PCM) is a toolkit for monitoring tge performance and energy metrics of Intel processors. [8] Both interfaces provide a very accurate estimation of the energy consumption.

B. Machine Learning Models

For our experiments we use three different models of feed-forward neural networks (FFNN) and two convolutional neural networks (CNN) that cover a representative set of machine learning applications. For the training of these models, we use real datasets (i.e., the Iris dataset [9], Mnist [10], and Cifar-10 [11]), as we describe in more detail below.

1) 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 [9] and even though it is one of the simplest feed-forward neural networks it can achieve high performance with an accuracy up to 97%.

2) Mnist-Small: The Mnist-Small is a feed-forward neural network that is based on the Mnist dataset [12], 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.

3) Mnist-Deep: The Mnist-Deep [13] 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.

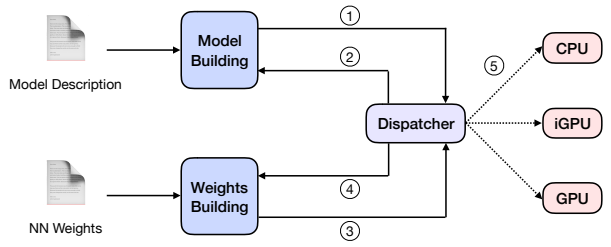


Fig. 2. The training phase of our system. The appropriate parameters of the neural networks are handled by the Model Building module, which builds the model (1) and passes it back to the Dispatcher module (2). Then, the weights of the neural network are passed to the Weights Building Module (3), which gives the resulted models and the corresponding weights back into the Dispatcher Module (4) that further loads them in each of the available processing devices (5).

4) Mnist-CNN: The Mnist-CNN is a model that has been also trained 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 2×2. Finally the FFNN has a dense layer of 128 nodes, leading to a 10-node output layer. The achieved accuracy of the model is 99.08%.

5) Cifar-10: The Cifar-10 is a convolutional neural network that has been trained on the Cifar-10 dataset [11], 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 $3\times3\times32$ and the pooling layer size is 2×2. Finally the FFNN has a dense layer of 128 nodes, leading to a 10-node output layer. The achieved accuracy of the model is 88%.

IV. Implementation

To achieve parallel classification across many devices we use OpenCL. In particular, we use the OpenCL implementation that comes with NVIDIA CUDA Toolkit 10.0, as well as the Intel OpenCL Runtime for the Core processor family. Our aim is to develop a portable implementation of the classification procedure of each of the models described in Section III-B, that can run efficiently on different, heterogeneous, devices.

A. Model Training

Machine learning algorithms usually build a mathematical model using a training process. There are many different ways for training; in this paper we use the following approach: For FFNNs, we pass the depth of the neural network, together with the number of nodes of each layer and the activation functions. For CNNs we also give the size and the number of filters of the convolutions, the size of the pooling, the corresponding activation functions and finally the description of the FFNN. As shown in Figure 2, all these are handled by the Model Building Module, which builds a model based on these information and passes it back to the Dispatcher module. The next step is to load the weights of the neural network to the main memory, based on the model that we have build. The Weights Building Module creates the appropriate buffers, loads the weights in the memory, and, finally, gives the buffers back into the Dispatcher Module. When the training phase completes, the resulted models and the corresponding weights are stored in the Dispatcher, which further loads them in each of the available processing devices.

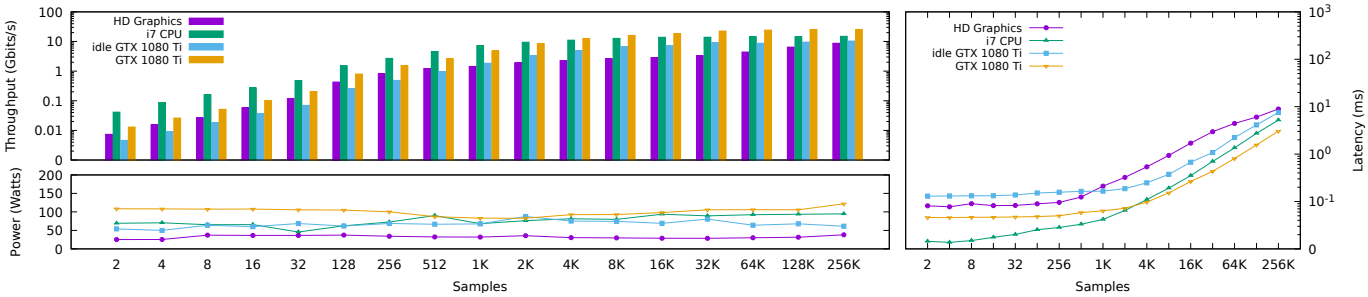
B. Parallelization

To execute the machine learning inference operations uniformly accross the different devices of the underlying system, we implement them on top of OpenCL 2.1. Our aim is to develop a portable implementation of each machine learning model, 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 procesor 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.

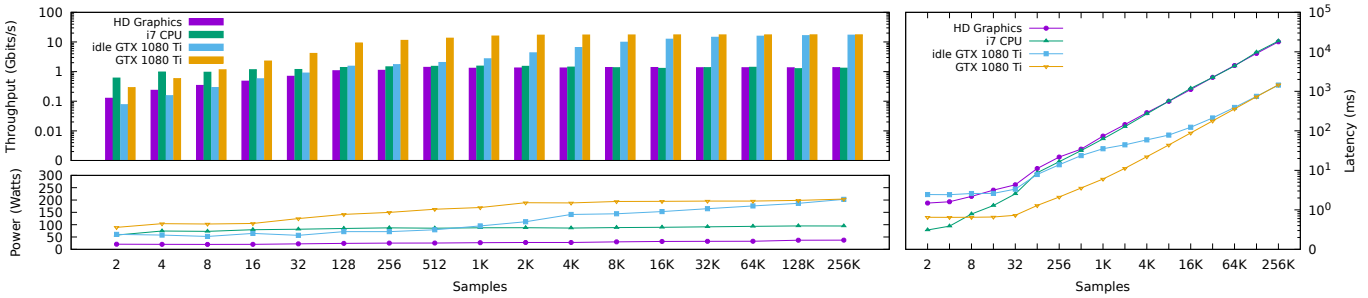
Overall, we have developed two different compute kernels, one for each type of neural networks. In OpenCL, an instance of a compute kernel is called a *work-item*; multiple work-items are grouped together and form *work-groups*. We follow a *thread-per-node* approach, and assign each work-item to handle (at least) one neural network layer; More specificaly, 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 architecures, like the Cifar-10 model.

The number of work-items per work-group affects significantly the performance of each device. For example, the GPUs require many and small work-groups, while CPUs prefer less and bigger work-groups. The reason for that is that the GPUs have a very fast thread scheduler that can hide memory-latencies, by scheduling other work-groups for execution. On the other hand, CPU 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 is 4096 work-items per work-group, whilst the best configuration for the GPU is 256 work-items per work-group, which at the same time is maximising the available registers per work-item.

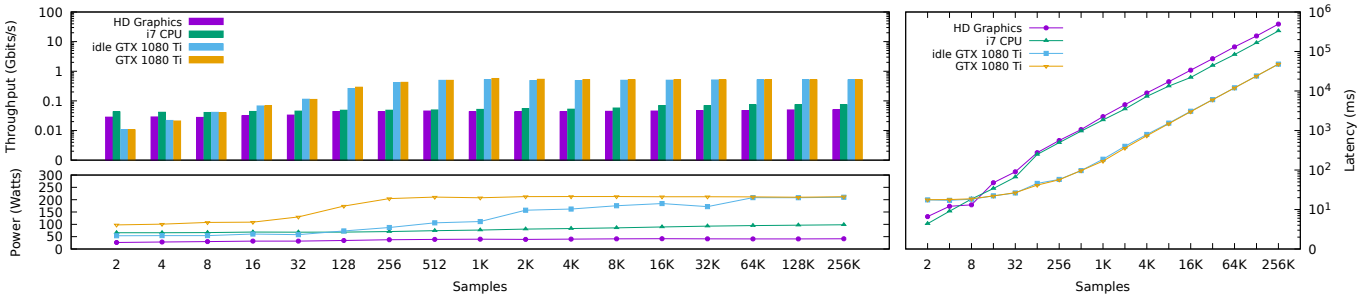
We notice that our devices enable heterogeneity on their memory model as well. For example the global memory of the GPU is physically independent of the memory of the host,



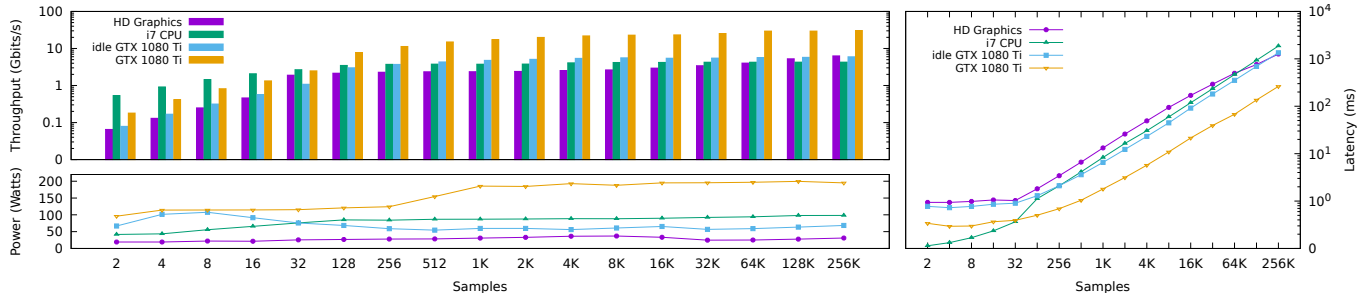
(a) Simple



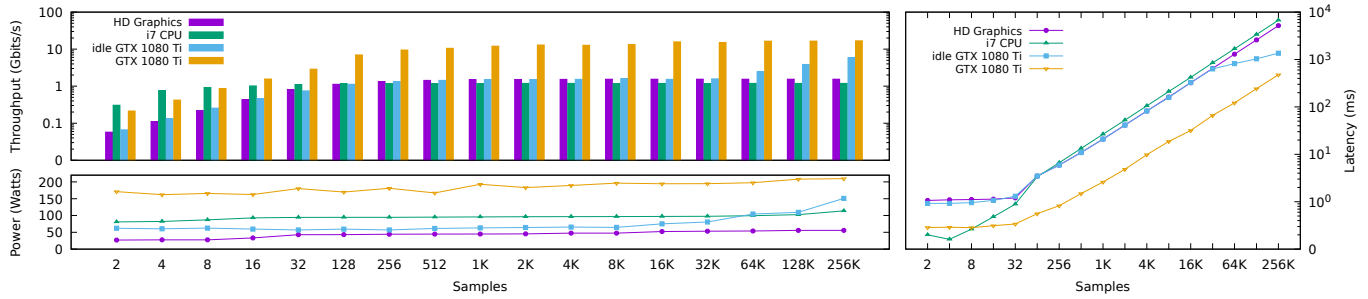
(b) Mnist Small



(c) Mnist Deep

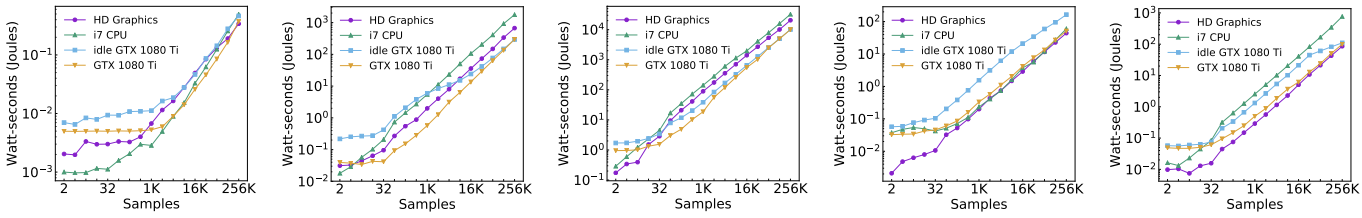


(d) Mnist CNN



(e) Cifar-10

Fig. 3. Throughput, latency and power consumption for each of the models presented in Section III-B



(a) Simple (b) Mnist Small (c) Mnist Deep (d) Mnist CNN (e) Cifar-10

Fig. 4. Watt-second (Joule) for each of the models presented in Section III-B

requiring a transfer of data from the host memory to the device's global memory through a PCIe transfer, for every computation. To avoid page swapping during the transfers, we copy the data that we want to classify in 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 in the global memory of each device, there is another critical aspect that affects the performance of our machine learning models. This 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 we found that 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 models. 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.

Finally, OpenCL provides a memory region, called local memory, that is shared by all work-items of a work-group. The local memory is implemented as an on-chip memory on GPUs, which is much faster than the off-chip global memory. Therefore, GPUs take advantage of local memory to improve performance. By contrast, CPUs do not have a special physical memory designed as local memory. As a result, all memory objects in local memory are mapped into sections of global memory, and will have a negative impact on performance. To overcome this, we explicitly stage data to local memory only when performing computations on the discrete GPU.

C. Performance Characterization

We now present the performance achieved by our machine learning models. Specifically, we measure the sustained *throughput*, *latency* and *power consumption* for each of the devices available in our base system. To accurately measure the energy required by 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.

Figure 3 summarises the results of our experiments for all the five models that we describe in Section III-B. 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 sample sizes [1]. We observe that the performance of all machine learning models 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 does also affect the sustained throughput in many of the machine learning models dramatically, with differences up to 7x. From all these observations, it is clear that in terms of throughput, no device performs best across all parameters. Instead, it is highly affected by the samples size, as well as the computational characteristics of the machine learning model, which in our case is the structure of the corresponding machine learning model. For example, Figure 3(a) shows 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 the sample sizes tested. In Figure 3(e), we observe that the CPU is better that the GPU for sample sizes up to 8 (when the

[1]In our measurements we have two options for the GTX 1080 Ti, namely *Idle GTX 1080 Ti* and *GTX 1080 Ti*. Nvidia uses Boost 3.0 tool to automatically adjust the GPU clocks in order to achieve better power consumption. In our measurements we have seen that the state of the GPU affects significantly the GPU performance, especially in the beginning of the measurements. Since it is not always feasible to control the state directly (especially in real-world setups), we provide both cases for the GPU: one when the GPU starts from an idle state and one when the GPU is warmed up.

GPU is warmed up); when the GPU starts from an idle state though, the CPU is better for sample sizes up to 128. Still, there are cases where the state of the GPU does not affect the performance, as we can see for example in Figure 3(c), where the CPU is better than the GPU for sample sizes up to 8, regardless of the starting state of the latter.

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 after a certain sample size, the latency grows linearly for all the machine learning models. This implies that the maximum throughput has been achieved. An exception can be observed in Figure 3(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 machine learning model 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 that for smaller sample sizes, the best achieved latency is affected by the state of it, specifically if the GPU is not warmed-up, then 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 3(d), we can see that given the GPU starts from an idle state, the best device is the CPU for sample sizes up to 256; when the GPU is warmed-up, the CPU is better for sample sizes up to 32.

Figure 4 shows the Joules that each device needs in order to perform the classification procedure for all the different machine learning models and different sample sizes. Due the heterogeneity of our system we can see that different devices perform better in different machine learning models and configurations; there is no device to rule them all. For example, even though the iGPU is the most power efficient device for all machine learning models (as shown in Figure 3), we get a different impression when we account the Joules consumed per device (depicted in Figure 4). 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 machine learning models than if it is warmed-up. We can also observe that the CPU is in many models the worst performing device. An increase in the sample size results to an increase in the consumed Joules, as it was expected, although we can observe that for each machine learning model from a sample size and above there is a linear increase in the consumption, which again indicates that the device has reached its maximum computational capacity. Still, each device reaches that point in a different sample size for each different model. For example, in Figure 4(b), the CPU reaches that point in sample size 1024, while the iGPU reaches that point for sample size equal to 512. As with the other metrics that we have analysed, the most appropriate device for a classification is changing based on the sample size in almost all the machine learning models. For example, in Figure 4(c) for the most appropriate device is the integrated GPU for sample sizes up to 8. However, for sample sizes of 16 and above, the most appropriate device is the GPU. We can see another example in Figure 4(b), where the state of the GPU affects significantly the appropriate device for the selected task. Specifically, for sample sizes between 8 and 4K, the iGPU is the most energy efficient device, if the GPU is not warmed-up, while the GPU is the most energy efficient device if it is warmed-up.

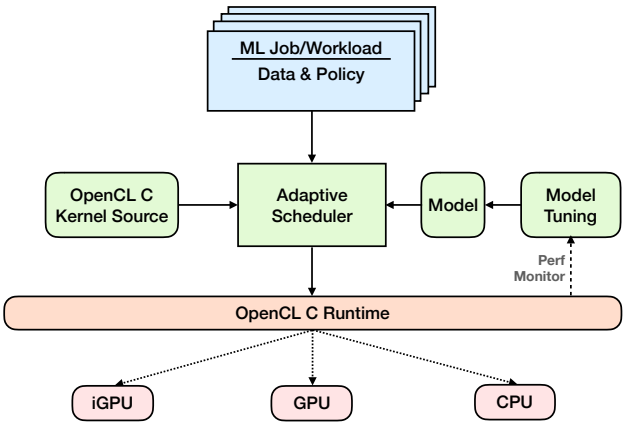


Fig. 5. The architecture of our adaptive scheduler. The scheduler loads a corresponding policy (i.e., lowest latency, energy efficiency or best throughput), alongside with the structure of the specified model, which uses to select the most appropriate device to perform the classification.

V. Efficient Device Selection via Scheduling

As we discuss in Section IV-C, the performance characterisation indicates that there is not a clear ranking between the benchmarked computational devices for executing machine learning inference operations. As a consequence of their architectural characteristics, some devices perform better under different performance metrics (e.g., throughput, latency, and power consumption), while these metrics may also deviate significantly among different applications. As an example, the CPU achieves the best performance on the Iris classification problem, but not on Mnist Deep classification problem.

It is obvious that our system favours heterogeneity in two different levels: *(i)* the different computational devices available and *(ii)* the diversified characteristics of the 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 maximise the performance or minimise the latency or energy consumption of our system.

A. Online Scheduler

The overview of our proposed scheduler is depicted in Figure 5. In essence, it reads the data from the input (e.g., network, file, or memory), alongside with the structure of the specified model and the corresponding policy (i.e., lowest latency, energy efficiency or best throughput). The scheduler also performs a PCIe call to check the state of the discrete

TABLE I
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}

GPU (idle or not). Based on these information, the scheduler determines the appropriate device to perform the classification.

At its core, the scheduler is based on machine learning 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.

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 VI, we present the evaluation results of the different models and corresponding tradeoffs.

Finally, we note that our scheduler is device-agnostic; even though we use a fixed set of processors and co-processors as described in Section III-A (which are representative though in a typical heterogeneous production system), our system can similarly operate when any other processors or co-processors are present (i.e., FPGAs, NPUs, or DSPs).

B. Data Augmentation and 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 IV-C, 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 III-B) 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.

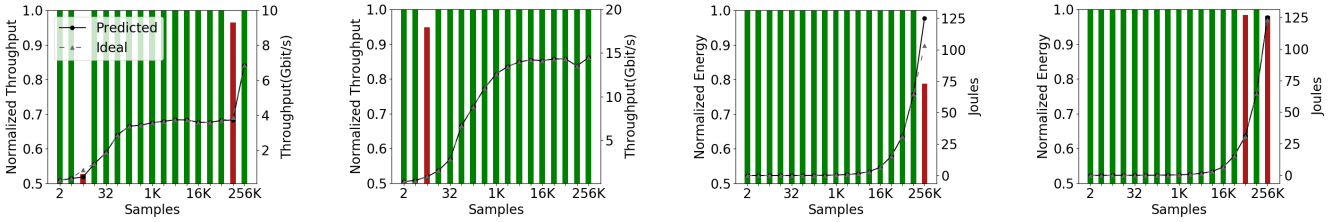
C. Training the Scheduler

It is well known that the Random Forests, usually, do not perform well on imbalanced data. Our approach to overcome this issue is twofold. Firstly, we do *not* evaluate its performance based on its accuracy, rather we report the F1-score that combines both precision and recall scores. Second, we perform a stratified k-Fold splitting to ensure that the classifier is trained with balanced data. More specifically, we do a Stratified k-Fold Nested cross validation to train our Random Forest classifier. The reason we apply Stratified k-Fold is due to the fact that the classes of our dataset are imbalanced; we do cross-validation to overcome the known overestimating problem of classifiers, and we do it in a nested way to find the best hyperparameters of the classifier.

Finally, we note that for the training of all the machine learning models we use the Scikit-learn programming framework on top of Python 3.6. Even though the nested cross validation is an iterative process, we can still parallelize the execution of each of the outer folds, as well as the inner folds. Each outer fold takes about 20 seconds to train in our base system, but due to the fact that all the jobs are running in parallel, the total training time of the Random Forest durates about 26 seconds. The hyperparameters that we tried are shown on Table I.

VI. EVALUATION

In this section, we evaluate our scheduler in terms of performance and accuracy. Table II shows the different machine learning models that the scheduler uses each time to make decisions. As we can see, these models have different benefits and tradeoffs that need to be carefully considered accordingly. For instance, Linear Regression provides the fastest inference execution times, while also requiring very small time for



(a) GPU is idle (b) GPU is warmed up (c) GPU is idle (d) GPU is warmed up

Fig. 6. Throughput achieved and energy consumed with the predictions of our scheduler

TABLE II
THE PERFORMANCE SUSTAINED BY OUR SCHEDULER FOR DIFFERENT MACHINE LEARNING MODELS

Model	Accuracy	Training Time	Classification Time
Baseline (Random Selection)	41%	N/A	0ms
Linear Regression	77.94%	15 sec	**0.7ms**
SVM	53.38%	2947sec	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

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 V-C, the F1-score correlates both precision and recall metrics, which gives a better understanding of how a model performs. Table III shows that Random Forest performs really well for scheduling decisions, 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 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(a) we can see that the scheduler made a wrong prediction for sample size 8 and the achieved 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.

TABLE III
SCHEDULER EFFICIENCY WHEN USING RANDOM FOREST CLASSIFIER

F1-score	Precision	Recall
93.51%	93.22%	93.21%

VII. RELATED WORK

a) Acceleration of machine learning applications.: Many works focus on accelerating the training and inference of machine learning applications. One such approach is the sparcification of DNNs [14], [15], [16] in which the inferece performance can be improved significantly, by decreassing the weights. Other approaches focus on decreasing the inference latency, by changing the DNN itself. For instance, binarized neural networks perform compression and pruning on the models to reduce memory consumption and computations for each inference [17]. Eyeriss [18] proposes a dataflow that exploits local data reuse and minimizes data movements in the neural network. All these approaches are orthogonal to our work and can be adapted as device-specific optimizations.

b) Heterogeneous processing environments.: The architectural heterogeneity in commodity computing systems has led many researchers to explore their abilities on different applications and workloads [19], [20], [21], [22], [23], [24], [25], [26], [27]. The performance ranking of different devices has been shown to have wide variations when executing different classes of network applications [19], [21], [22]. To capture the performance variability effect, it is important to perform in-field studies and quantitative evaluation of the different processing units [26], [27]. Besides that, a heterogeneous processing system that employs CPUs and GPUs has to solve many challenges, including the distribution of the workload on processors with different capabilities and the data transfers bottleneck [23].

c) Performance prediction.: The utilization of external GPUs has been long used for the acceleration of a wide set of applications [28], [29], [30], [31], [32], [33], [34], [35], [36], [37], [38], [39]. Instead of using the GPUs exclusively, some works focus on predicting their performance and utilize them opportunistically. For instance, in [40], 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 predicting the performance of the devices, using compiler-based and neural network approaches accordingly [41], [42]. Lastly, in [43] they try to predict the performance of heterogeneous systems using machine learning models and the hardware specifications of each device. A major difference 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.

VIII. Conclusion

In this work we address the problem of improving the efficiency of machine learning classification on commodity, off-the-self, CPU-GPU architectures. In particular, 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.

Acknowledgements

This work was supported by the projects CONCORDIA, C4IIoT, COLLABS, and MARVEL funded by the European Commission under Grant Agreements No. 830927, No. 833828, No. 871518, and No. 957337. This publication reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

References

[1] Rajat Raina, Anand Madhavan, and Andrew Y. Ng. Large-Scale Deep Unsupervised Learning Using Graphics Processors. ICML, 2009.

[2] James Bergstra, Frédéric Bastien, Olivier Breuleux, Pascal Lamblin, Razvan Pascanu, Olivier Delalleau, Guillaume Desjardins, David Warde-Farley, Ian Goodfellow, and Arnaud Bergeron. Theano: Deep learning on gpus with python. NIPS, 2011.

[3] Leyuan Wang, Zhi Chen, Yizhi Liu, Yao Wang, Lianmin Zheng, Mu Li, and Yida Wang. A Unified Optimization Approach for CNN Model Inference on Integrated GPUs. ICPP, 2019.

[4] N. Ho and W. Wong. Exploiting half precision arithmetic in Nvidia GPUs. In *HPEC*, 2017.

[5] A. Dhakal and K. K. Ramakrishnan. NetML: An NFV Platform with Efficient Support for Machine Learning Applications. In *NetSoft*, 2019.

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

[7] NVIDIA System Management Interface. Available: http://developer.download.nvidia.com/compute/DCGM/docs/nvidia-smi-367.38.pdf.

[8] Intel Performance Counter Monitor - A Better Way to Measure CPU Utilization. Available: https://software.intel.com/content/www/us/en/develop/articles/intel-performance-counter-monitor.html.

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

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

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

[12] 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.

[13] 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.

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

[15] 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.

[16] 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.

[17] 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.

[18] Y. Chen, J. Emer, and V. Sze. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. In *ISCA*, 2016.

[19] Lazaros Koromilas, Giorgos Vasiliadis, Ioannis Manousakis, and Sotiris Ioannidis. Efficient Software Packet Processing on Heterogeneous and Asymmetric Hardware Architectures. In *ACM/IEEE ANCS*, 2014.

[20] Joongi Kim, Keon Jang, Keunhong Lee, Sangwook Ma, Junhyun Shim, and Sue Moon. NBA (network balancing act): a high-performance packet processing framework for heterogeneous processors. In *EuroSys*. ACM, 2015.

[21] Eva Papadogiannaki, Lazaros Koromilas, Giorgos Vasiliadis, and Sotiris Ioannidis. Efficient Software Packet Processing on Heterogeneous and Asymmetric Hardware Architectures. *IEEE/ACM Trans. Netw.*, 25(3):1593–1606, jun 2017.

[22] Giannis Giakoumakis, Eva Papadogiannaki, Giorgos Vasiliadis, and Sotiris Ioannidis. Pythia: Scheduling of Concurrent Network Packet Processing Applications on Heterogeneous Devices. In *6th IEEE NetSoft*, pages 145–149, 2020.

[23] Viktor Rosenfeld, Sebastian Breß, and Volker Markl. Query Processing on Heterogeneous CPU/GPU Systems. *ACM Comput. Surv.*, 55(1), 2022.

[24] Dimitris Deyannis, Lazaros Koromilas, Giorgos Vasiliadis, Elias Athanasopoulos, and Sotiris Ioannidis. Flying Memcache: Lessons Learned from Different Acceleration Strategies. In *IEEE SBAC-PAD*, 2014.

[25] Kai Zhang, Kaibo Wang, Yuan Yuan, Lei Guo, Rubao Li, Xiaodong Zhang, Bingsheng He, Jiayu Hu, and Bei Hua. A Distributed In-Memory Key-Value Store System on Heterogeneous CPU—GPU Cluster. *The VLDB Journal*, 26(5):729–750, oct 2017.

[26] Carole-Jean Wu, David Brooks, Kevin Chen, Douglas Chen, Sy Choudhury, Marat Dukhan, Kim Hazelwood, Eldad Isaac, Yangqing Jia, Bill Jia, Tommer Leyvand, Hao Lu, Yang Lu, Lin Qiao, Brandon Reagen, Joe Spisak, Fei Sun, Andrew Tulloch, Peter Vajda, Xiaodong Wang, Yanghan Wang, Bram Wasti, Yiming Wu, Ran Xian, Sungjoo Yoo, and Peizhao Zhang. Machine Learning at Facebook: Understanding Inference at the Edge. In *IEEE HPCA*, 2019.

[27] Siqi Wang, Anuj Pathania, and Tulika Mitra. Neural Network Inference on Mobile SoCs. *IEEE Design Test*, 37(5):50–57, 2020.

[28] Giorgos Vasiliadis, Spiros Antonatos, Michalis Polychronakis, Evangelos P. Markatos, and Sotiris Ioannidis. Gnort: High Performance Network Intrusion Detection Using Graphics Processors. In *RAID*, 2008.

[29] Giorgos Vasiliadis, Michalis Polychronakis, Spiros Antonatos, Evangelos P. Markatos, and Sotiris Ioannidis. Regular Expression Matching on Graphics Hardware for Intrusion Detection. In *RAID*, 2009.

[30] Giorgos Vasiliadis and Sotiris Ioannidis. GrAVity: A massively parallel antivirus engine. In *RAID*, September 2010.

[31] Giorgos Vasiliadis, Michalis Polychronakis, and Sotiris Ioannidis. MIDeA: A Multi-Parallel Intrusion Detection Architecture. In *ACM CCS*, 2011.

[32] Keon Jang, Sangjin Han, Seungyeop Han, Sue Moon, and KyoungSoo Park. SSLShader: Cheap SSL Acceleration with Commodity Processors. In *NSDI*, 2011.

[33] Giorgos Vasiliadis, Michalis Polychronakis, and Sotiris Ioannidis. Parallelization and characterization of pattern matching using GPUs. In *IEEE IISWC*, 2011.

[34] Peter Bakkum and Kevin Skadron. Accelerating SQL database operations on a GPU with CUDA. In *Proceedings of the 3rd workshop on general-purpose computation on graphics processing units*, pages 94–103, 2010.

[35] Pramod Bhatotia and Rodrigo Rodrigues. Shredder: GPU-Accelerated Incremental Storage and Computation. In *USENIX FAST*, 2012.

[36] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. GPUfs: Integrating a file system with GPUs. *ACM Transactions on Computer Systems (TOCS)*, 32(1):1–31, 2014.

[37] Evangelos Ladakis, Giorgos Vasiliadis, Michalis Polychronakis, Sotiris Ioannidis, and Georgios Portokalidis. GPU-Disasm: A GPU-based x86 Disassembler. In *ISC*, 2015.

[38] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[39] Giorgos Vasiliadis, Lazaros Koromilas, Michalis Polychronakis, and Sotiris Ioannidis. Design and implementation of a stateful network packet processing framework for GPUs. *IEEE/ACM Transactions on Networking*, 25(1):610–623, 2016.

[40] I. Baldini, S. J. Fink, and E. Altman. Predicting GPU Performance from CPU Runs Using Machine Learning. In *SBAC-PAD*, 2014.

[41] 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 *IISWC*, 2017.

[42] Klaus Kofler, Ivan Grasso, Biagio Cosenza, and Thomas Fahringer. An automatic input-sensitive approach for heterogeneous task partitioning. In *ICS*, 2013.

[43] 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 *NCA*, 2016.