Transparent Acceleration of Java-based Deep Learning Engines

Preprint · September 2020

DOI: 10.1145/3426182.3426188

Athanasios Stratikopoulos
The University of Manchester
11 PUBLICATIONS **18** CITATIONS
SEE PROFILE

Mihai-Cristian Olteanu
The University of Manchester
1 PUBLICATION **0** CITATIONS
SEE PROFILE

Zoran Sevarac
13 PUBLICATIONS **102** CITATIONS
SEE PROFILE

Nikos Foutris
The University of Manchester
13 PUBLICATIONS **50** CITATIONS
SEE PROFILE

Transparent Acceleration of Java-based Deep Learning Engines[*]

Athanasios Stratikopoulos
athanasios.stratikopoulos@manchester.ac.uk
The University of Manchester
Manchester, United Kingdom

Mihai-Cristian Olteanu
mihaicristian.olteanu98@gmail.com
The University of Manchester
Manchester, United Kingdom

Ian Vaughan
ian.vaughan@student.manchester.ac.uk
The University of Manchester
Manchester, United Kingdom

Zoran Sevarac
zoran.sevarac@deepnetts.com
Deep Netts LLC
Belgrade, Serbia

Nikos Foutris
nikos.foutris@manchester.ac.uk
The University of Manchester
Manchester, United Kingdom

Juan Fumero
juanfumero@acm.org
The University of Manchester
Manchester, United Kingdom

Christos Kotselidis
christos.kotselidis@manchester.ac.uk
The University of Manchester
Manchester, United Kingdom

ABSTRACT

The advent of modern cloud services, along with the huge volume of data produced on a daily basis, have increased the demand for fast and efficient data processing. This demand is common among numerous application domains, such as deep learning, data mining, and computer vision. In recent years, hardware accelerators have been employed as a means to meet this demand, due to the high parallelism that these applications exhibit. Although this approach can yield high performance, the development of new deep learning neural networks on heterogeneous hardware requires a steep learning curve. The main reason is that existing deep learning engines support the static compilation of the accelerated code, that can be accessed via wrapper calls from a wide range of managed programming languages (e.g., Java, Python, Scala). Therefore, the development of high-performance neural network architectures is fragmented between programming models, thereby forcing developers to manually specialize the code for heterogeneous execution. The specialization of the applications' code for heterogeneous execution is not a trivial task, as it requires developers to have hardware expertise and use a low-level programming language, such as OpenCL, CUDA or High Level Synthesis (HLS) tools.

In this paper we showcase how we have employed TornadoVM, a state-of-the-art heterogeneous programming framework to transparently accelerate Deep Netts on heterogeneous hardware. Our work shows how a pure Java-based deep learning neural network engine can be dynamically compiled at runtime and specialized for particular hardware accelerators, without requiring developers to employ any low-level programming framework typically used for such devices. Our preliminary results show up to 6.45x end-to-end performance speedup and up to 88.5x kernel performance speedup, when executing the feed forward process of the network's training on the GPUs against the sequential execution of the original Deep Netts framework.

[*]Work-in-Progress paper.

CCS CONCEPTS

• **Software and its engineering → Software development techniques**.

KEYWORDS

Deep Learning, Hardware Acceleration, Java

ACM Reference Format:
Athanasios Stratikopoulos, Mihai-Cristian Olteanu, Ian Vaughan, Zoran Sevarac, Nikos Foutris, Juan Fumero, and Christos Kotselidis. 2020. Transparent Acceleration of Java-based Deep Learning Engines. In *International Conference on Managed Programming Languages & Runtimes (MPLR '20), November 4–6, 2020, Virtual Event, United Kingdom.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3426182.3426188

1 INTRODUCTION

Artificial intelligence is continually gaining popularity with its main objective being to enable computers to make decisions that are normally made by domain experts. This domain is a superset

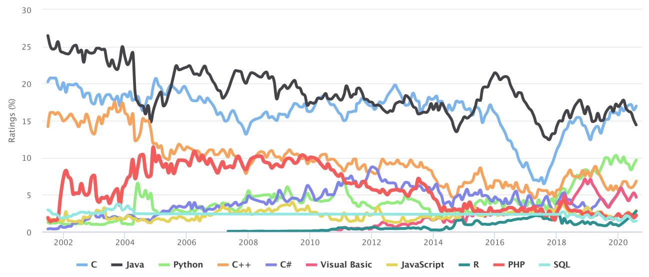

Figure 1: The TIOBE index of the most common programming languages in August 2020. Source: www.tiobe.com.

of Machine Learning and Deep Learning, which both rely on training a mathematical model by using historical data related to the application of interest.

To program machine learning and deep learning applications, developers tend to use high-level programming languages (e.g., Java, Python, etc.), due to the ease of programming and maintenance, and invoke pre-compiled kernels for hardware acceleration. Figure 1 presents a ranking graph with all programming languages based on the popularity in the "TIOBE Index for August 2020". As shown in Figure 1, Java has been the most popular language among all programming languages since 2002; with a small interpolation with the C language during the last decade.

The proliferation of heterogeneous systems has resulted in the introduction of numerous, novel programming frameworks [2, 13], with the vast majority of them supporting the C or the C++ programming languages. These frameworks support the generation of code for GPU execution via CUDA or OpenCL, or FPGA execution via High Level Synthesis (HLS), and lately through OpenCL. However, the portability across diverse hardware devices comes at the cost of programmability, as developers are required to have hardware-specific knowledge about the underlying accelerators. The execution of Java programs (and managed languages in general) onto heterogeneous accelerators is an active research topic [11, 12] with many challenges associated with the compilation, the memory management and the runtime system, which all lie at the core of the managed languages. In this case, a runtime system (e.g., the Java Virtual Machine -JVM-) can make use of the hardware resources without the developers' intervention, thereby enabling the applications to exploit the traditional philosophy of Java: *write once, run everywhere* from a heterogeneous acceleration perspective. Some of those frameworks are Marawacc [6, 9], FastR-GPU [8], Aparapi [3], JaBEE [17], IBM GPU J9 [10] and TornadoVM [7].

The acceleration of machine learning or deep learning applications is applicable through frameworks, such as TensorFlow [2] and PyTorch [13] for Python or Deep-learning4j [4] for Java. These frameworks rely on existing pre-compiled code that can be accessed via wrapper calls from a wide range of managed programming languages (e.g., Java, Python, Scala). However, in most cases the accelerated pre-compiled code is optimized for a specific hardware accelerator and cannot be ported to any other device type. To enable portability or implement new neural network features for hardware acceleration, developers must be acquainted with the low-level programming frameworks (e.g., CUDA, OpenCL), as well as extending the wrapper calls. On the other hand, a radical different approach includes the automatic acceleration of the deep learning applications onto any heterogeneous hardware resource, such as GPU, FPGA or multi-core CPU.

In this paper, we present our work towards mitigating the gap in programmability, while enabling users to accelerate the deep learning frameworks directly from Java. To achieve that, we have employed TornadoVM [7] to accelerate a pure Java-based deep learning engine such as Deep Netts [16]. In detail ,this paper makes the following contributions:

- Analyses the Deep Netts framework to identify potential code segments for acceleration.
- Applies the TornadoVM API to parallelize the original Deep Netts framework.
- Evaluates the performance of the proposed Tornado-Deep Netts against the original Deep Netts implementation, showcasing performance speedups of up to 6.45x and 88.5x for the end-to-end and the kernel executions, respectively.

2 DEEP LEARNING OVERVIEW

Deep learning is a field that has recently emerged and it uses the structure of an artificial neural network (ANN) which comprises multiple layers of artificial neurons, in order to train a model to autonomously perform a task, such as visual recognition, speech recognition, etc. [14]. This structure is first trained using some application-dependent data and then uses the learned model to take intelligent decisions or perform accurate predictions. The main core in an ANN is the artificial neuron historically known as `perceptron`. An artificial neuron performs three main tasks:

- It accepts some input values either from the data or from the outputs of neurons, depending on the layer in which this neuron belongs to.
- It applies some weights and biases to produce an intermediate value known as the `net input` value. These parameters are learned during the training of the network so as to increase the accuracy.
- The *net input* value is forwarded to the `activation function`, which performs a non-linear transformation.

The most widely used model of ANNs consists of multiple artificial neurons grouped into three types of layers: *input layer*, the *hidden layers*, and the *output layer*. The input layer accepts as input the features from the dataset, and each neuron in this layer can be connected with one or more neurons in the next hidden layer, the outputs of which are connected with the next hidden layer and so on, until the connections reach the output layer. Then, the output layer uses a cost/loss function as a metric to assess the accuracy of the model. If the accuracy is not satisfactory, then the ANN uses a method to automatically learn internal representations [15]. One of the most widely accepted methods is the backward pass phase, which uses a gradient descent approach to update the parameters in each layer so that the average loss decreases. This ANN is also called as `feed forward neural network` as the flow of the information beyond the layers of the network flows forward. Other types of neural networks are the `convolutional` and the `recurrent` neural networks. For the rest of this paper we will focus on the feed forward neural network model, which is applicable to a wide range of domains.

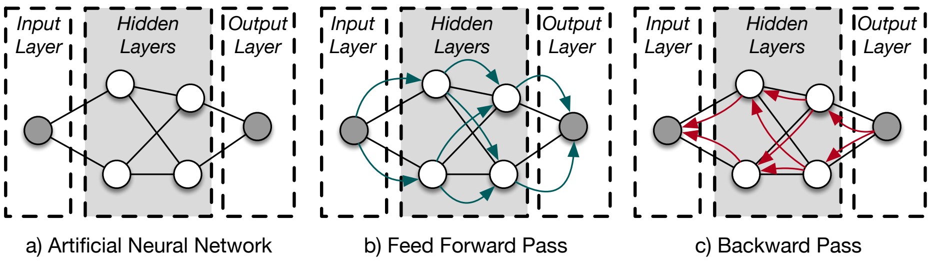

Figure 2: The primary steps of training in Deep Netts. Figure 2a depicts a fully connected artificial neural network that contains two hidden layers, each of which encloses two neurons. Figure 2b shows the feed forward pass which includes all activations values from each neuron in the network. Figure 2c performs the backward pass that adjusts the configuration of each neuron in order to obtain a highly effective training.

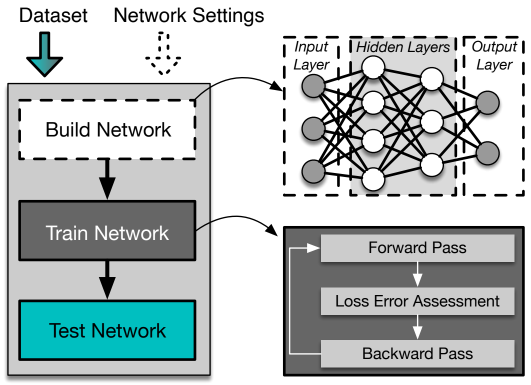

Figure 3: The work-flow of Deep Netts.

2.1 Deep Netts Deep Learning Engine

Deep Netts [16] is a deep learning development platform that offers a deep learning library and an integrated development tool. As it is entirely implemented in Java, it enables Java developers to seamlessly apply deep learning in their applications. Currently, Deep Netts supports dominant supervised learning algorithms, including feed-forward and convolutional neural networks. The supported types of neural network layers are: the *fully connected*, the *convolutional*, the *maximum pooling*, and the *softmax output* layer. The key features and advantages of Deep Netts include:

- Ease of use, thanks to beginner and Java developer friendly API.
- The integration of the state-of-the-art models of neural networks, such as the feed forward network and the convolutional network. These types of networks are provided out-of-box and require less understanding of background theory and library internals in order to be effectively used.
- The provision of advanced visualization and logging for understanding, debugging and solving data-based, architecture-based, or training-based, issues [16].
- Portability, ease of integration, distribution and maintenance (thanks to pure Java implementation), which are features of great importance for large scale deployments.
- Deep Netts is a base for reference implementation of standard Java API for visual recognition and machine learning JSR 381 [1], which is being developed within the official Java technology standardization organization.

Nonetheless, one of the main disadvantages of Deep Netts compared to other libraries, is that it lacks support for GPU/FPGA acceleration, so training with large amounts of images, or big images can be time consuming. However, the pure Java implementation, along with the clean design and readable code make it suitable for experiments which can evolve Java platform towards better support for deep learning and machine learning in general.

2.1.1 *The Deep Netts Work-flow.*

Figure 3 illustrates the main work-flow in Deep Netts for building neural networks, and training them to run corresponding algorithms. As shown in Figure 3, Deep Netts accepts as input the data set upon which the training will be based along with various configuration parameters for the network. These parameters can be the type and the number of layers, and the maximum error rate that the algorithm can tolerate. Once the inputs are set, the next step is to build the artificial neural networks and initialize the values of weights and biases in each neuron of every layer. Figure 2a presents a simplistic view of a fully connected built neural network. The next process in the workflow is the training which comprises three parts: the forward pass, the loss error assessment, and the backward pass.

(1) The **forward pass** triggers the activation functions of each neuron in each layer and creates an activation value which is forwarded to the next layer, until it reaches the output layer. This process is presented in Figure 2b with the green arrows. The complexity of the feed forward process can be significant, as it is in accordance with the structure of the neural network. Section 4 will discuss in detail how we parallelized this step for the *fully connected* layer.
(2) The **loss error assessment** checks the emerged output from the previous part and makes a comparison with the configured maximum error value. In case the network has not met the error level (accuracy) that is required, the backward pass is performed.
(3) The **backward pass** traverses all the layers in the reversed order going backwards. This process is responsible for updating the weights and biases in each neuron of the layers in order to increase the overall performance of the model. Figure 2c represents this process with arrows depicted in red. This part of the training is considered as computationally expensive and can merit to be executed in parallel [14].

3 TORNADOVM PROGRAMMING FRAMEWORK

TornadoVM is a plugin to OpenJDK that allows Java programmers to automatically execute their applications on heterogeneous hardware. Currently, TornadoVM can accelerate Java programs on multi-core CPUs, GPUs and FPGAs. Additionally, TornadoVM can migrate, at runtime, execution from one device to another [7] (e.g., from a multi-core CPU to a GPU).

Figure 4 shows the three main TornadoVM components (i.e., API, Runtime, Compiler), along with the execution engine which is responsible for the Just In Time (JIT) compilation of the bytecodes and the memory management. TornadoVM exposes a lightweight API that developers use to indicate which Java methods they would like TornadoVM to accelerate on heterogeneous devices. Once the user identifies the methods, TornadoVM compiles, at run-time, Java bytecodes to OpenCL C as follows: a) it builds a data-flow graph with the aim to optimize the data dependencies between tasks, and subsequently reduce the required data transfers and buffer allocation time; b) it generates new bytecodes (TornadoVM Bytecodes) on top of the Java bytecodes, which are used for pure orchestration on the heterogeneous devices; and c) it executes the whole application by using the TornadoVM bytecode, and it compiles at runtime, the input Java methods to OpenCL C code [5].

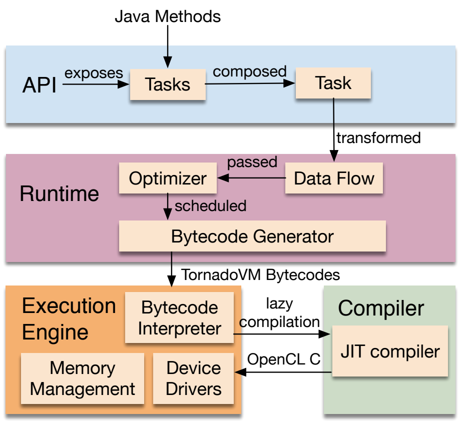

Figure 4: The TornadoVM work-flow.

TornadoVM API:. Since this work uses the TornadoVM API to accelerate neural networks written in Java onto heterogeneous devices, we present in this section an example of how TornadoVM is programmed. TornadoVM exposes an API that expresses the task-based parallelism, in which each task is a reference to an existing Java method. Additionally, the TornadoVM API can create a group of tasks that can be compiled together in the same *compilation unit*, and subsequently run on the same target device (e.g., the same GPU). This group of tasks is called a `TaskSchedule`.

Developers programming with TornadoVM can also annotate the code by using the `@Parallel` annotation that informs the TornadoVM JIT compiler that a loop is a candidate for parallel execution. Moreover, there is the `@Reduce` annotation that informs the compiler for a reduction operation, in which an input array is reduced to a scalar value computed with an associative and commutative operator. The basic characteristic of the TornadoVM API is that it allows Java programmers to exploit hardware parallelism without requiring the knowledge of OpenCL or hardware architecture. As part of the fall-back execution mechnanism, the annotated code is adapted based on the characteristics of the device, and in case of single-threaded CPU execution it can be ignored.

Listing 1: Java code-snippet to run matrix multiplication by using TornadoVM.

```java
1  public class Compute {
2    private static void mxm(Matrix2D A, Matrix2D B, Matrix2D C) {
3      for (@Parallel int i = 0; i < SIZE; i++) {
4        for (@Parallel int j = 0; j < SIZE; j++) {
5          float sum = 0.0f;
6          for (int k = 0; k < SIZE; k++)
7            sum += A.get(i, k) * B.get(k, j);
8          C.set(i, j, sum);
9    }}}
10   public void run(Matrix2D A, Matrix2D B, Matrix2D C) {
11     TaskSchedule t = new TaskSchedule("s0")
12       .task("t0", Compute::mxm, matrixA, matrixB, matrixC)
13       .streamOut(matrixC).execute();
14   }}
```

Listing 1 shows an example for computing a matrix multiplication by using the TornadoVM API. Lines 2-8 show the code of the method that encompasses the sequential implementation of the matrix multiplication. Lines 3 and 4 have been enhanced by using the `@Parallel` annotation in order to hint the TornadoVM compiler that these loops can be parallelized. Then, the method `run` (line 10) instantiates a `TaskSchedule` object with a single task, pointing to the `mxm` method. Finally, the code is executed in line 13. Note that the Java code is totally agnostic about the hardware device on which the program will be executed. Once the program is annotated, the TornadoVM runtime compiles and executes all tasks enclosed within the `TaskSchedule`.

4 HOW IS DEEP NETTS PARALLELIZED WITH TORNADOVM?

This section presents how we extended Deep Netts to use TornadoVM for the training phase of the **forward pass** for the *fully connected* layer. The parallel execution of Deep Netts onto heterogeneous accelerators is an ongoing work, therefore we also plan to parallelize the backward pass phase in a similar way as the forward pass, as well as extending to other layers in the neural network.

Listing 2 presents the code snippet of the forward method in Deep Netts, whereas Listing 3 shows the transformations that were applied to the `forward_sequential` method. At first, we needed to convert the Java object types to primitive arrays. Those primitive arrays were used as method parameters to the `forward_tornado` method in lines 1-2 of Listing 3. The second transformation was the addition of the `@Parallel` annotation in line 3, as discussed in Section 3. This annotation is used as a hint by the TornadoVM compiler to parallelize those loops with OpenCL.

The last transformation that we did was to indicate to TornadoVM which method to offload onto the GPU. Listing 4 presents the the `TaskSchedule` object that we used. We first declared the arrays to copy into the target device and passed them as input to the `streamIn` method (line 2). Then we declared a task (line 3), which points to the `forward_tornado` method. Finally, we added the output arrays that should be synchronized with the host (the main CPU) after the execution as input to the `streamOut` method, and then we called the `execute` method (line 5).

Listing 2: The original source code of the forward method in Deep Netts.

```java
private void forward_sequential() {
  for (int i = 0; i < outputs.getCols(); i++) {
    for (int j = 0; inCol < inputs.getCols(); j++)
        outputs.add(i, inputs.get(j) * weights.get(j, i));
        float val = getActivationValue(outputs, i);
        outputs.set(i, val));
}}
```

Listing 3: TaskSchedule that builds the forward task from the FullyConnectedLayer class with TornadoVM.

```java
private void forward_tornado(float[] inputs, float[] weights,
                             float[] outputs) {
  for(@Parallel int i = 0; i < outputs.length; i++) {
    for (int j = 0; j < inputs.length; j++)
      outputs[i] += inputs[j] * weights[getIndex(j, i)];
    float val = getActivationValue(outputs, i);
    outputs[i] = val;
}}
```

Listing 4: TaskSchedule that builds the forward task from the FullyConnectedLayer class with TornadoVM.

```java
1  TaskSchedule forwardTask = new TaskSchedule("DNets")
2    .streamIn(inputs, weights)
3    .task("forward", FullyConnectedLayer::forward_tornado,
4         inputs, weights)
5    .streamOut(outputs).execute();
```

4.1 Benefits of Our Approach

The main benefits of our approach against the state-of-the-art deep learning frameworks are as follows:

(1) Developers can seamlessly accelerate any method of the deep learning framework. On the contrary, other frameworks (e.g., TensorFlow [2]) are capable of accelerating only the methods for which a respective pre-compiled binary exists.
(2) A developer can transparently deploy and execute a Java method onto any hardware device, spanning from multi-core CPUs, to discrete and integrated GPUs and FPGAs.
(3) The execution of a deep learning framework that follows our approach can be migrated from a GPU to an FPGA, or even another GPU at runtime. This is a key feature for dealing with trade-offs regarding performance and power dissipation, or other runtime factors, such as device availability; that many of the existing frameworks lack of.

5 EXPERIMENTAL EVALUATION

To evaluate the Tornado-Deep Netts implementation against the original Deep Netts[1] implementation, we used two classes of GPUs (AMD, Nvidia), along with the same operating system and heap size (Table 1) to execute the benchmark applications presented in Table 2. We performed the warm-up process which included at least 200000 executions prior to the actual timing of both systems, in order to warm up the JVM and fairly compare both frameworks.

[1]We used the Deep Netts community edition.

Table 1: The experimental hardware and software characteristics of our testbed.

CPU	Intel Core i7-8700K CPU @ 3.70GHz
Memory	64 GB
JVM	OpenJDK 1.8.0_201 64-bits
JVM Heap Size	16 GB
GPUs	AMD Radeon RX Vega 64 GPU Nvidia GeForce GTX 1050 Ti GPU
Operating System	Ubuntu 18.04.01
GCC Compiler	v7.4.0

The reported results in Section 5.2 are the median of the last 10 iterations of each measurement with respect to the JVM variance.

5.1 Benchmark Applications

In our experiments we evaluated the performance of the forward method in the *fully connected* layer, as presented in Section 4. To assess the performance comparison of the *fully connected* layer on both systems (Tornado-Deep Netts and Deep Netts), we used six applications from the Deep Netts application suite that utilize this layer for training their neural network. Table 2 presents four applications, along with the list of hidden layers that are used for training. *BostonHouses* is an application that predicts the Boston housing prices. *CreditCard* detects unusual fraud patterns of credit card transactions. *SpamClassifier* is a binary classifier for spam classification. *IrisClassifier* classifies flowers into one of the possible categories, based on several input-defined flower features.

As shown in Table 2, we used the *tanh* as activation function of the hidden layers, while the *linear*, *sigmoid* and *softmax* were used as the activation functions of the output layer. Additionally, the *Mean Squared Error* and the *Cross Entropy* were used as the loss functions. Finally, Table 2 shows the workload distribution which corresponds to the width size of the layer. The width size is equal to the number of neurons in the layer. Depending on the application characteristics, the size of the width can impact the correctness of the training model. Hence, Table 2 presents the width sizes that have resulted in the correct functionality of the training model in both systems.

5.2 Performance Analysis

As Deep Netts is exclusively implemented in Java and the Community Edition is executed on a single CPU thread, it has focused on small neural networks that can efficiently execute on single-threaded Java. To assess the benefits of GPU hardware, we iterated our experiment across different width sizes of the *fully connected* layer.

Figure 5 presents the end-to-end performance speedup of the Tornado-Deep Netts system against the original Deep Netts system for a wide range of width sizes. As shown in Figure 5, small and medium workloads perform better on the original Deep Netts implementation which is entirely executed through the host JVM on the CPU. This performance penalty is attributed to the dispatching of the OpenCL commands by the GPU driver, the execution of the TornadoVM Bytecodes at runtime, and the data transfers from the main memory to the GPU memory, and backwards. In particular, the dispatching can take up to 31%, 28% and 21% (*SpamClassifier*

Table 2: Benchmark applications. This table presents the hidden layers used in the neural network of each benchmark, along with the respective activation function per layer (hidden layers/output layer). The next column presents the loss function used in each benchmark. The workload distribution presented in the last column is partitioned in small, medium and large.

Benchmarks	Hidden Layers	Activation Function		Loss Function	Workload		
		Hidden Layers	Output Layer		Low	Medium	Large
BostonHouses	FullyConnected	tanh	linear	Mean Squared Error	64	512	2048
CreditCard	FullyConnected	tanh	sigmoid	Cross Entropy	64	512	8192
SpamClassifier	FullyConnected	tanh	sigmoid	Cross Entropy	64	512	4096
IrisClassifier	FullyConnected	tanh	softmax	Cross Entropy	64	512	16384

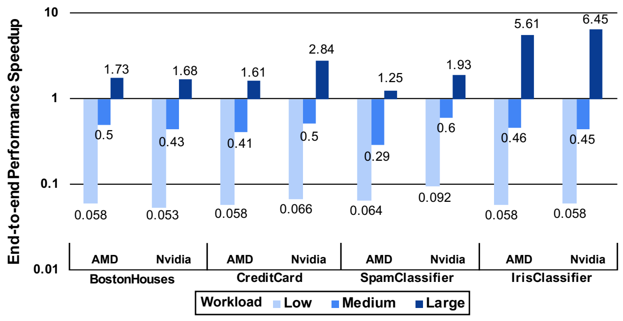

Figure 5: The end-to-end performance speedup of the fully connected layer running through the Tornado-Deep Netts (GPU) against the Original Deep Netts (single-threaded CPU) for various width sizes (small, medium, large).

Benchmark - Nvidia GPU) of the end-to-end execution time for small, medium and large workloads, respectively. Additionally, the data transfers can take up to 30%, 36% and 45% (*SpamClassifier Benchmark - Nvidia GPU*) of the end-to-end execution time for small, medium and large workloads, while the time spent for the execution of the TornadoVM bytecodes takes up to 15% (small workload), 15% (medium workload) and 27% (large workload) (*SpamClassifier Benchmark - Nvidia GPU*), accordingly. Thus, for small and medium workloads these three parts consume more time than the actual execution of the kernel. On the other hand, for large workloads in which the computation exceeds that overhead, Tornado-Deep Netts achieves performance speedup against Deep Netts ranging from 1.25x (*SpamClassifier Benchmark - AMD GPU*) up to 6.45x (*IrisClassifier Benchmark - Nvidia GPU*).

Unlike Figure 5, Figure 6 presents the performance comparison of the kernel time of Tornado-Deep Netts against Deep Netts when executing on two GPU cards for small, medium and large workloads. This figure confirms that for small workloads, and in some cases including the medium workloads (*CreditCard Benchmark - AMD GPU*), the actual computation of the `forward` method is not worthy to be performed on the GPU. On the other hand, for the majority of medium and large workloads, the kernels offloaded on the GPU can exhibit performance speedups, ranging from 1.8x (*IrisClassifier Benchmarks - AMD GPU*) up to 88.5x (*IrisClassifier Benchmark - Nvidia GPU*). The performance difference (up to 90% for *CreditCard Benchmark* - large workload) of the kernels executed on the AMD and the Nvidia GPUs are due to the behavior of the OpenCL driver on the AMD GPU, which performs data transfers when running the kernels. Thus, the column bars for the AMD GPU in Figure 6 include the data transfers, and the difference between the AMD

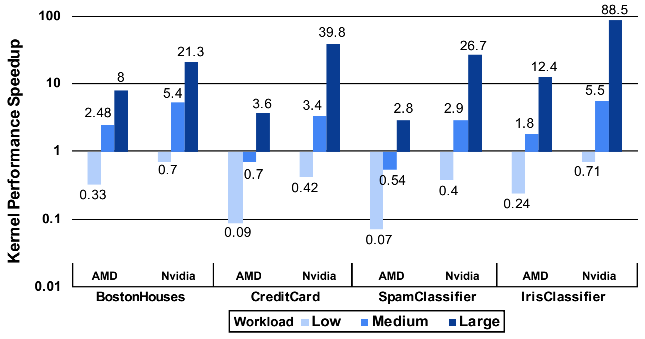

Figure 6: The kernel performance speedup of the fully connected layer running through the Tornado-Deep Netts (GPU) against the Original Deep Netts (single-threaded CPU) for various width sizes (small, medium, large).

bars in Figure 5 and Figure 6 is due to the time taken by the driver for dispatching the OpenCL commands and the time for executing the TornadoVM Bytecodes. Although our work is still in progress, our results show that Java-based deep learning engines can be combined with JIT compilers (e.g., TornadoVM) to transparently run on heterogeneous hardware, without requiring any hand-crafted OpenCL or CUDA kernels.

6 CONCLUSIONS AND FUTURE WORK

This paper presents our work in progress towards transparently accelerating deep learning engines written in Java on heterogeneous systems. To showcase our approach we used Deep Netts, a deep learning engine fully implemented in Java that lacks support for heterogenous hardware. Then we employed TornadoVM, a state-of-the-art heterogeneous programming framework that compiles and executes Java applications onto OpenCL-compatible devices, without requiring any significant knowledge about hardware. Our preliminary results showed that the automatically generated OpenCL kernels running on GPUs can outperform the original Java methods by up to 6.45x and up to 88.5x for end-to-end and kernel execution, respectively.

As future work, we plan to accelerate the remaining layers of Deep Netts (e.g., convolutional layer, max pooling layer, etc.), and conduct experiments on FPGAs.

ACKNOWLEDGMENTS

This work is partially supported by the EU Horizon 2020 E2Data 780245 grant. We would like to thank Foivos Zakkak and Konstantinos Papangelou for their feedback.

REFERENCES

[1] [n.d.]. Java JSR 381: Visual Recognition (VisRec) Specification. https://jcp.org/en/jsr/detail?id=381

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*.

[3] AMD. [n.d.]. Aparapi project. https://github.com/aparapi/aparapi

[4] D. Alex Black, Adam Gibson, and Josh Patterson. 2017. Deeplearning4j. https://deeplearning4j.org/

[5] James Clarkson, Juan Fumero, Michail Papadimitriou, Foivos S. Zakkak, Maria Xekalaki, Christos Kotselidis, and Mikel Luján. 2018. Exploiting High-Performance Heterogeneous Hardware for Java Programs Using Graal. In *Proceedings of the 15th International Conference on Managed Languages and Runtimes (ManLang'18)*. Association for Computing Machinery, New York, NY, USA, Article 4, 13 pages. https://doi.org/10.1145/3237009.3237016

[6] Juan Fumero. 2017. *Accelerating Interpreted Programming Languages on GPUs with Just-In-Time and Runtime Optimisations*. Ph.D. Dissertation. The University of Edinburgh, UK.

[7] Juan Fumero, Michail Papadimitriou, Foivos Zakkak, Maria Xekalaki, James Clarkson, and Christos Kotselidis. 2019. Dynamic Application Reconfiguration on Heterogeneous Hardware. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*.

[8] Juan Fumero, Michel Steuwer, Lukas Stadler, and Christophe Dubach. 2017. Just-In-Time GPU Compilation for Interpreted Languages with Partial Evaluation. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Xi'an, China) *(VEE'17)*. Association for Computing Machinery, New York, NY, USA, 60–73. https://doi.org/10.1145/3050748.3050761

[9] Juan José Fumero, Toomas Remmelg, Michel Steuwer, and Christophe Dubach. 2015. Runtime Code Generation and Data Management for Heterogeneous Computing in Java. In *Proceedings of the Principles and Practices of Programming on The Java Platform* (Melbourne, FL, USA) *(PPPJ'15)*. Association for Computing Machinery, New York, NY, USA, 16–26. https://doi.org/10.1145/2807426.2807428

[10] K. Ishizaki, A. Hayashi, G. Koblents, and V. Sarkar. 2015. Compiling and Optimizing Java 8 Programs for GPU Execution. In *International Conference on Parallel Architecture and Compilation (PACT)*.

[11] Christos Kotselidis, James Clarkson, Andrey Rodchenko, Andy Nisbet, John Mawer, and Mikel Luján. 2017. Heterogeneous Managed Runtime Systems: A Computer Vision Case Study. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '17)*. https://doi.org/10.1145/3050748.3050764

[12] Christos Kotselidis, Sotiris Diamantopoulos, Orestis Akrivopoulos, Viktor Rosenfeld, Katerina Doka, Hazeef Mohammed, Georgios Mylonas, Vassilis Spitadakis, and Will Morgan. 2020. Efficient Compilation and Execution of JVM-Based Data Processing Frameworks on Heterogeneous Co-Processors. In *Proceedings of the 23rd Conference on Design, Automation and Test in Europe (DATE '20)*.

[13] 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. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc.

[14] Josh Patterson and Adam Gibson. 2017. *Deep Learning: A Practitioner's Approach* (1st ed.). O'Reilly Media, Inc.

[15] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. 1985. *Learning internal representations by error propagation*. Technical Report. California Univ San Diego La Jolla Inst for Cognitive Science.

[16] Zoran Sevarac. 2018. Deep Netts Betta. https://deepnetts.com/

[17] Wojciech Zaremba, Yuan Lin, and Vinod Grover. 2012. JaBEE: Framework for Object-oriented Java Bytecode Compilation and Execution on Graphics Processor Units. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*.