This paper has been accepted for presentation in the 5th Workshop on Accelerated Machine Learning (AccML), co-located with HiPEAC'23.

HEP-BNN: A Framework for Finding Low-Latency Execution Configurations of BNNs on Heterogeneous Multiprocessor Platforms

Leonard David Bereholschi, Ching-Chi Lin, Mikail Yayla, Jian-Jia Chen
Technical University of Dortmund, Germany
{leonard.bereholschi, chingchi.lin, mikail.yayla, jian-jia.chen}@tu-dortmund.de

***Abstract*—Binarized Neural Networks (BNNs) significantly reduce the computation and memory demands with binarized weights and activations compared to full-precision NNs. Executing a layer in a BNN on different devices of a heterogeneous multiprocessor platform consisting of CPU and GPU can affect the inference performance, i.e., accuracy and latency. Usually, a heterogeneous HW platform consisting of a CPU and a GPU is available to execute the BNN workloads. However, to use the heterogeneous HW effectively, it is necessary to find an efficient strategy for BNN workload mapping. In this work, we propose a framework that generates efficient BNN layer-to-device mappings (i.e. suitable parallel configuration for each layer of the model) for execution platforms comprised of CPU and CUDA-capable GPU. We evaluate our proposed framework with two BNN architectures using two well-known datasets, *Fashion-MNIST* and *CIFAR-10*, on three hardware platforms with different characteristics. The results show that compared to running a fully-parallelized GPU implementation, our framework generates an efficient configuration up to $2\times$, $2.6\times$ and $11.8\times$ faster on our tested hardware respectively.**

***Index Terms*—Binarized Neural Network, inference, GPU, CUDA**

I. Introduction

Neural Networks (NN) have been applied to various practical domains in the last decade, e.g., image recognition in computer vision, prediction of chemical patterns in chemistry, and cancer detection in medical science. [1] Given a well-trained NN model, *inference* is the process of using the model to make predictions against previously unseen data. Depending on the structure of the NN model, the inference can be time- and resource-consuming.

Binarized Neural Network (BNN) [2] is a resource-efficient variant of NNs. In BNNs, the weights and the activations are binarized into 1-bit representation. Multiplications and accumulations in a BNN can be computed using the *xnor* operand and *popcount()*, respectively. Therefore, BNNs are significantly more resource-efficient compared to full-precision NNs, which makes BNNs excellent candidates for running AI application on resource-constrained edge devices.

For executing BNN workloads, customized hardware accelerators (i.e. on FPGAs or ASICs) are still in the early stages of development [3], while *CPUs and GPUs are mature and readily available*. Several recent studies have evaluated the use of GPUs for BNNs. Hubara et al. [2] evaluate BNNs using *XNOR* kernels for GPUs. Xu et al. [4] implement a computation kernel for BNNs as well. Li et al. [5] run BNNs on Turing GPUs, focusing on the bit-level parallelism and strides in memory. Chen et al. [6] develop a BNN acceleration engine for mobile phones.

Although GPU provides high computing capability, executing every layer in a BNN on GPU does not guarantee good performance. We reveal in this work that executing all the BNN layers *exclusively* on the GPU leads to significant increase in latency compared to running the model on the CPU sequentially (e.g. model is too small and CPU-overhead too significant, see Fig. 1). Therefore, a proper layer-to-device mapping is imperative for achieving efficient BNN inference on heterogeneous multiprocessor platforms consisting of CPU and GPU.

Contributions: In this paper, we propose a framework, *HEP-BNN*, which automatically generates an efficient layer-to-device mapping (i.e. the suitable parallel configuration for each layer) for a given BNN model running on a heterogeneous multiprocessor platform consisting of CPU and GPU. Given a trained BNN model, *HEP-BNN* systematically evaluates the execution time of the model on CPU and on GPU under different parallel configurations. The configuration with the least execution time is highlighted as the efficient CPU/GPU configuration for the BNN on target platform. Such a framework would lead to a more efficient use of available hardware resources. Furthermore, automatically generating directly applicable code containing the optimized mapping, would enable highly efficient BNN inference. It would allow both researchers and practitioners to fully exploit the capabilities of their available hardware platforms in applying BNNs efficiently on resource-limited edge devices.

Our contributions are summarized as follows:

- We present our *HEP-BNN* framework that automatically generates the efficient layer-to-device mapping for given BNN models and heterogeneous execution platforms consisting of a CPU and a CUDA-capable GPU. The generated code, containing the efficient configuration for each layer of the model, can then be used for applications using BNN inference in practice. The proposed framework is published on Github[1].

[1] https://github.com/LeonardDavid/hep-bnn

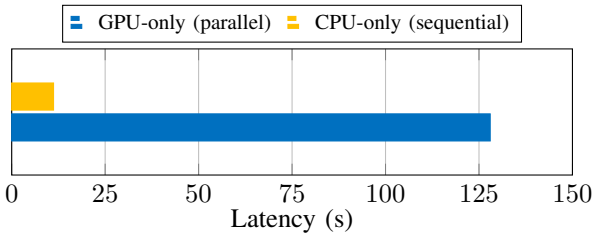

Fig. 1. Fashion-MNIST on Jetson TX2: example from out results for the difference of total latency between the sequential CPU model and the parallel GPU model (with a higher CPU-overhead).

- To demonstrate the capabilities of our framework, we apply our framework on two BNN models with two commonly used datasets, *Fashion-MNIST* and *CIFAR10*, on three hardware platforms with different characteristics: *Server*, *Laptop*, and *Jetson TX2*. Across the different BNN models, our results show that by applying properly chosen parallel configurations for layers running on GPU, inference times can be reduced by up to $2\times$, $2.6\times$ and $11.8\times$ for the three execution platforms, respectively.

II. System Model

In Section II-A, the basics of BNNs the layers used in our models are presented. An introduction into GPU computing is given in Section II-B, where the CUDA framework is also described, including its features, limitations, and some implementation details. The different parallel configurations implemented and used in our framework, as well as their notations used throughout this paper, are presented in Section II-C. Finally, the problem definition is elaborated in Section II-D.

A. Basics of Binarized Neural Networks

Binarized Neural Networks (BNNs) [2] are a resource-efficient variant of NNs. In a BNN model, the weights and the activations are binarized into 1-bit representation. Unlike the full-precision NNs, where one matrix multiplication must be performed for computing the output of each neuron, we can simply apply $xnor$ operands for computing the outputs of neurons in BNNs. Specifically, the output of a layer can be computed with

$$2 * popcount(xnor(W_i^l, I^{l-1})) - \#bits > T,$$

where W_i^l are the weights of layer l, I^{l-1} are the inputs to layer l, $popcount()$ is a function which counts the number of 1s in the results of the $xnor$ operand, $\#bits$ is the number of bits in the $xnor$ operand, and T is a learnable threshold parameter which can be computed with the batch normalization parameters. The binary outputs depend on the truth value of the statement, which represents a shifted binarization function [7].

In this work, we consider convolutional BNNs. There are four basic types of layers in a convolutional BNN, i.e., *convolutional* layer, *maxpool* layer, *step* layer, and *fully-connected* layer, as shown in Figure 2. We also employ the *flattening* layer in the BNNs in this work.

The *convolutional* layer computes a 2D convolution of the input with filters. We use "Cχ" to denote a *convolutional* layer, where χ is a number indicating the amount of neurons in the layer. For example, "C64" is a *convolutional* layer with 64 neurons. In this work, the filter size is fixed at 3×3 for all the *convolutional* layers.

The *maxpool* layer downsamples the input by selecting the maximum value of the input in a given window size, which is set to 2×2 in our models. We use "MPχ" to denote a *maxpool* layer, where χ indicates it's output size, e.g., "MP16" for a output of size "16×16".

A *step* layer performs batch normalization followed by a binary activation function. Batch normalization [7] is used in NN models for faster and more stable training, which helps increase the accuracy. Note that the threshold values in the batch normalization are still signed integers, even in BNNs. In our models, we apply Hard-Tanh as our binary activation function. For inference in a BNN, batch normalization followed by activation can be computed with binary thresholding [7]. We denote the *step* layer as "S" in the rest of this paper.

A *fully-connected* layer connects every neuron in the current layer with every neuron in the next layer. We denote a *fully-connected* layer as "FCχ", where χ is the amount of neurons in the layer. Note that *fully-connected* layers also have binary weights as learnable parameters.

We use "FLAT" to denote a *flattening* layer. The layer rearranges a high dimension matrix into a lower dimension matrix, e.g., from a 3-dimension matrix into a 1-dimension array in our models. The rearrangement can be done with a simple one-line operation on CPU in C++ code.

B. GPU

Due to their architecture, GPUs are suitable for highly parallel use cases, such as repetitive matrix multiplication for graphics-intensive tasks. Most state-of-the-art GPUs have numerous computation cores, operating in an efficient manner with large and fast memories. On GPUs, the computations are performed by threads in parallel. To program the thread behaviours, specialized GPU code, e.g., frameworks such as CUDA or OpenCL, needs to be employed. In our work, we use the CUDA programming language to deploy the computation workload of a BNN model on Nvidia graphics cards.

In a CUDA program, threads can be arranged into *thread blocks*, in which up to 1024 threads are executed in parallel. Thread blocks can be further organized into a 3D *grid* structure, allowing for a greater degree of parallelization on the GPU. The size of the grid is limited on the different dimension axes as follows: $\{x, y, z\} \rightarrow \{2^{31}-1, 65536, 65536\}$. Furthermore, the usage of internal CUDA variables, such as *threadIDx* and *blockIDx*, allow each thread to address individual values from arrays related to its specific computation.

Functions in which computation tasks are executed on GPU are called *kernels* in the CUDA programming language. Before launching a *CUDA kernel*, memory is allocated on the GPU memory, after which all the required data for the computations is transferred from the host (CPU) to the device (GPU). The sizes of the thread blocks and the grid are also specified before the *kernel* launch, while respecting previously

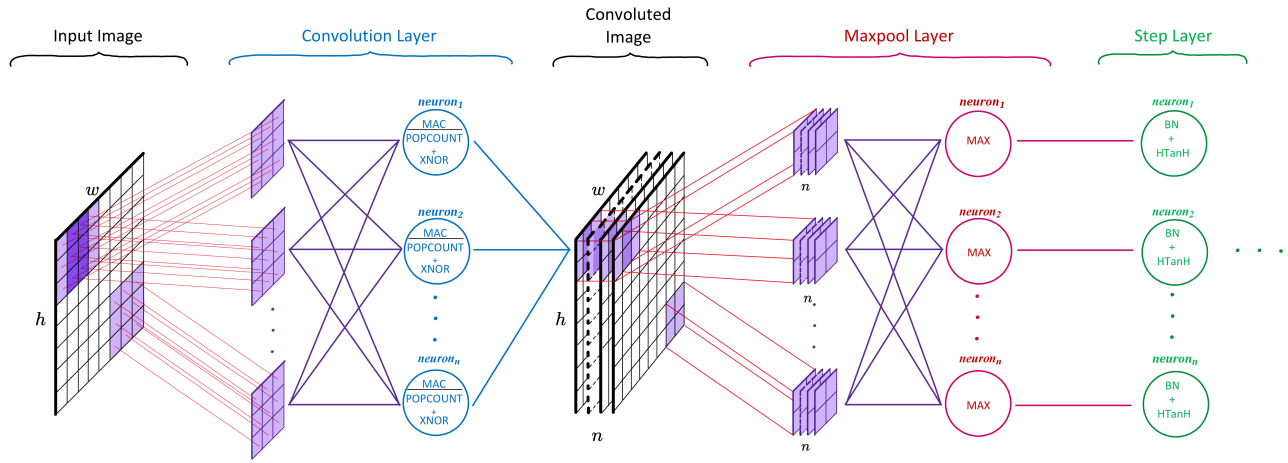

Fig. 2. Structure of a Convolutional BNN model demonstrating the three major layer types: *Convolution*, *Maxpool*, and *Step* layer.

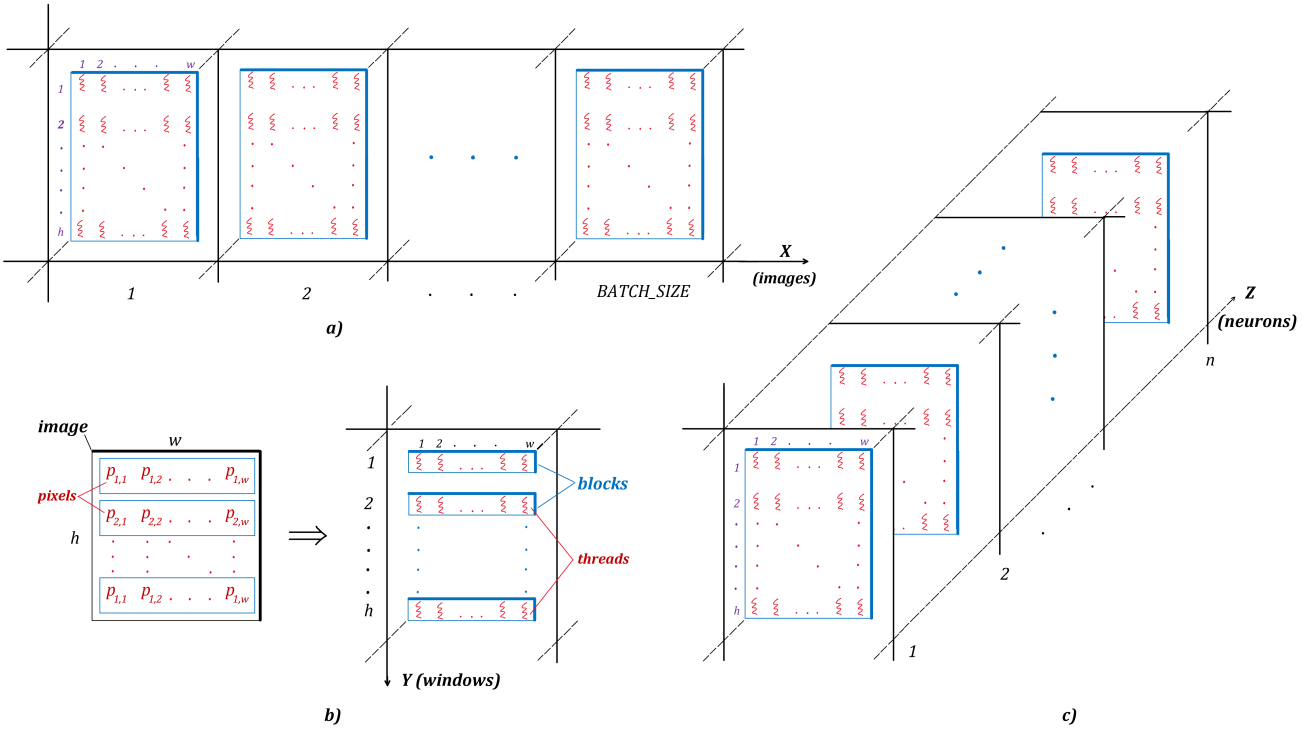

Fig. 3. Concepts of the parallelism aspects: a) *Data-based*, b) *Window-based*, and c) *Neuron-based*

mentioned limitations. After the GPU finishes executing every task in the *kernel*, the results are copied from the device back to the host, and the previously allocated memory on the GPU is freed.

Although GPUs provide massive computational power compared to CPU, and are often used as accelerators in many use cases, running an application on the GPU does not always lead to performance improvement. In some cases, running parallel code on GPU can take longer than the sequential CPU code because of, for example, time overheads in communication. Therefore, an analysis on the characteristics of an application, e.g., degree of parallelism, can determine if it is beneficial to run the application on GPU. For applications that can be accelerated by GPU, how to organize the workload to achieve the optimal performance is a crucial issue that needs to be solved.

C. Data Parallelism Aspects

The workload of a BNN model consists of multiple data images which are used as input. We define *batch size* as the amount of data images in a batch, that are processed concurrently. To process the workload of BNN inference on a data set in parallel, we organize the workloads based on three aspects of data parallelism:

1) *Data-based*: every data image in a batch is inferred concurrently.
2) *Window-based*: a data image is divided into convolution windows of consecutive pixels, with the windows being processed concurrently.
3) *Neuron-based*: the outputs of the neurons in the same layer in a NN model are calculated concurrently.

In the *Data (X)* configuration, multiple data images in a batch are inferred on the GPU concurrently, as shown in Figure 3 (a). Each data image in a batch is assigned to one

GPU thread block. If a thread block is assigned with multiple data images, these images are processed one after another. Each pixel (and its subsequent operations) in a data image is processed by one thread in the thread block.

Figure 3 (b) demonstrates the idea for the *Window (Y)* configuration, in which a data image is divided into windows of consecutive pixels in a row-wise manner, with each window being assigned to one GPU thread block. The workloads related to the pixels (threads) in the same window are processed on the GPU concurrently.

For the *Neuron (Z)* configuration, the outputs of neurons from the same layer are calculated concurrently, as shown in Figure 3 (c). The output of a neuron is the weighted sum from its predecessors after going through an activation function. Each neuron in a layer is assigned to a GPU thread block, with threads in a thread block taking the corresponding outputs from the previous layer as input, and calculating the output for the neuron.

Using these aspects, we consider the following seven parallel configurations and their notations, which will be used throughout this paper: 1) *Data (X)*, 2) *Window (Y)*, 3) *Neuron (Z)*, 4) *Data + Window (XY)*, 5) *Data + Neuron (XZ)*, 6) *Window + Neuron (YZ)*, and 7) *Data + Window + Neuron (XYZ)*. The configurations composed of multiple aspects are implemented according to all of the implementations of the individual aspects at the same time. Note that for all the parallel configurations, the threads in thread blocks perform the same operation depending on the layer, e.g., convolution of pixels in a convolution layer. The *blockIDx* and *threadIDx* variables determine which pixel(s) and/or neuron(s) each thread is responsible for.

D. Problem Definition

Given a well-trained BNN model, we aim to reduce the inference time of a BNN model with the help of GPU. However, there are multiple aspects for parallelizing the computation workloads on GPU. Each layer in the BNN model can have different suitable parallel configurations. Nevertheless, there might also be layers with workloads that are not beneficial if running on GPU due to overheads, e.g., which are caused by data migration between host and the GPU device. Therefore, our objective is to generate an efficient layer-to-device mapping for a given BNN model, so that the inference time is minimized. For layers that are mapped to GPU for execution, we also determine their suitable parallel configurations.

III. Framework Presentation

We introduce the proposed framework in details in this section. The operational steps of our *HEP-BNN* framework are outlined in Section III-A. The mapping algorithm is described in Section III-B. Information about the folder structure, important script files, and generated files of the framework, are detailed in Sections III-C, III-D and III-E, respectively.

A. High-level Overview of Our HEP-BNN Framework

The operational steps performed by our framework are represented in Figure 4. First, the program receives a BNN model in ONNX format as input, previously trained on a specific dataset (e.g. Fashion-MNIST, CIFAR10). Then, for every batch size in a defined range, the appropriate C++ and CUDA code for the CPU and GPU is generated. After every layer $(L_1, \ldots, L_n)$ is implemented using different configurations, the model is inferred for every configuration applied, which results in different runtimes (with $t_{11}, \ldots, t_{1m} \in L_1$ and $t_{n1}, \ldots, t_{nm} \in L_n$). These timing information are used for choosing an efficient configuration in a greedy manner, according to Alg. 1 described in Section III-B.

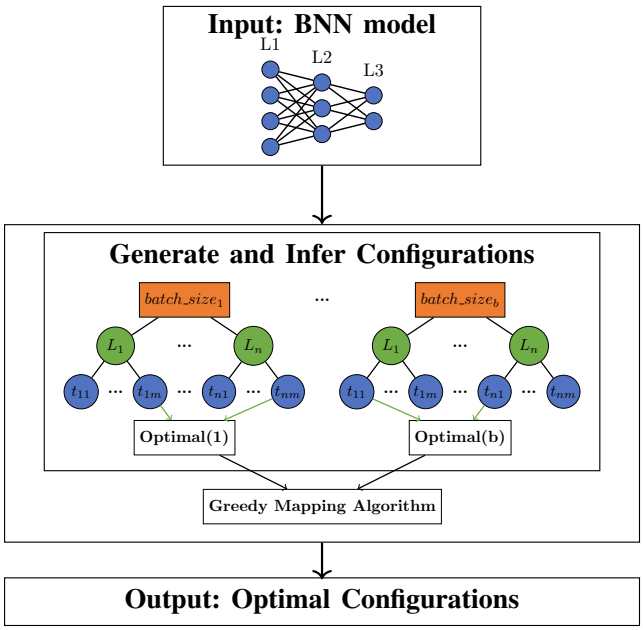

Fig. 4. Operational steps our *HEP-BNN* framework.

B. Mapping Algorithm

In order to determine the suitable *parallel configuration* which achieves the lowest inference time, the following layer-to-device mapping algorithm is applied (see Alg. 1). The algorithm profiles each layer of the BNN model using different batch sizes, both on the CPU and the GPU. On the GPU, every parallel configuration from Section II-C are implemented. In total, each layer is profiled on 8 different implementations: 1) *CPU*, 2) *Data (X)*, 3) *Window (Y)*, 4) *Neuron (Z)*, 5) *Data + Window (XY)*, 6) *Data + Neuron (XZ)*, 7) *Window + Neuron (YZ)*, and 8) *Data + Window + Neuron (XYZ)*.

The *implementation* that achieves the lowest inference time for the profiled *layer* is mapped to the specific *batch size*. Summing up the lowest inference times for every *layer*, results in the total runtime for executing the BNN model, using the *efficient implementations* for the *specific batch size*.

After profiling every *layer*, the **minimal** total runtime, as well as the *proper batch size* for which it is achieved, is searched. Finally, the *proper batch size* is used to get the mapped *implementation* of each *layer* of the BNN model. This creates the *efficient parallel configuration*, which achieves the lowest expected inference time. Algorithm 1 represents the pseudocode of the described mapping algorithm.

Data: BNN model
Result: Efficient Configuration

1 $proper_{batch\_size} \leftarrow 0$
2 $result\_time \leftarrow$ MAX_INT
3 **foreach** $batch\_size$ **do**
4 $sum_{min\_time} \leftarrow 0$
5 **foreach** $layer$ **do**
6 $min\_time \leftarrow$ MAX_INT
7 **foreach** $implem$ **do**
8 implement $layer$ using $implem$
9 (CPU_{time}, GPU_{time}) $\leftarrow$ profile(implemented$_{layer}(batch_size)$)
10 $inference\_time \leftarrow \text{CPU}_{time} + \text{GPU}_{time}$
11 **if** $inference\_time < min\_time$ **then**
12 $min\_time \leftarrow inference\_time$
13 MAP $implem(layer)$ to $batch\_size$
14 **end**
15 **end**
16 $sum_{min\_time} \leftarrow sum_{min\_time} + min\_time$
17 **end**
18 **if** $sum_{min\_time} < result\_time$ **then**
19 $result\_time \leftarrow sum_{min\_time}$
20 $proper_{batch\_size} \leftarrow batch\_size$
21 **end**
22 **end**
23 **foreach** $layer$ **do**
24 get $implem(layer)$ from MAP[$proper_{batch\_size}$]
25 add $layer_{implem}$ to *Efficient Configuration*
26 **end**
27 **return** *Efficient Configuration*

Algorithm 1: Mapping algorithm that determines the efficient configuration of a BNN model that achieves the lowest inference time (Note: *implem* is short for *implementation*)

C. Folder Structure

Our *HEP-BNN* framework uses Python to run the implementations and optimizations of the input model. To exemplify our implementation, we use the open source machine learning compiler and code generator Fastinference [8]. Specifically, for the generation of the C++ and CUDA code for the CPU and GPU respectively, the templating language *Jinja2* is used. An overview of the most important part of the folder-tree structure (*'fastinference/'*) will be outlined in this section.

Each model, optimizer and implementation is defined in a separate folder in *'fastinference/'*. These are further separated into the supported algorithm types such as *'ensemble/'*, *'tree/'*, *'neuralnet/'*. Specifically, in *'fastinference/implementations/neuralnet'* there are folders for the different target hardware: *'cpp/'*, *'fpga/'*, *'iree/'*.

This is where the main part of our work is located, namely in the *'cuda/'* folder, which contains a separate directory for each parallel configuration. The folder names follow the notations introduced in Section II-C.

In every folder there are *Jinja2* files containing mainly CUDA code templates for every layer, for parallel execution of the model on the GPU. There is also a *'cpu/'* folder, that contains C++ templates for the sequential operation of the model on the CPU, used for the sequential implementations. Each *'implement.py'* file found in every folder, contains the function *'to_implementation()'* and is responsible for selecting the appropriate template files for each layer of the model, and to generate the necessary C++ and CUDA files for the implementation.

Additionally, the *'automatic/'* folder contains the code which runs our mapping algorithm, that automatically reads the model and data from the appropriate path, generates and infers all of the selected configurations, and maps the suitable configuration in a greedy manner. In the following section, we give a more detailed description of the usage of certain files, by using the CIFAR10 dataset as an example.

D. Important script files

In ***test_cuda.py***, the implementations of the parallel configurations which can be used, are specified. Their notations are consistent with the ones described in Section II-C. Afterwards, the *HEP-BNN* framework is launched by calling the *'to_implementation()'* function found in ***test_utils.py***.

Here, an upper and lower bound is set for the batch sizes, which are expressed as powers of 2 (e.g. with 'b_l = 0' and 'b_u = 4', the batch sizes used are $2^0 = 1$, $2^1 = 2$, $2^2 = 4$, and $2^3 = 8$).

The mapping algorithm is then run inside of the nested for-loops, one for each configuration, and the other one for each batch size. It consists of two important functions, *'prepare_fastinference()'* and *'run_experiment()'*. The former generates all the necessary files for the model to compile and run (more details will be given in section III-E), while the latter function compiles and runs the generated model, after which it outputs and stores the results of the inference.

After running all of the generated models, the best configuration for each layer is mapped according to the lowest inference time (see Alg. 1 – lines 23:26). Finally, the efficient configuration is generated and inferred, achieving the lowest inference time out of all other combinations.

The BNN model is stored in ONNX format under the following path:

```
fastinference/implementations/
neuralnet/cuda/automatic/model/
cifar10/model_cifar10.onnx
```

Test data is stored under:

```
fastinference/implementations/
neuralnet/cuda/automatic/data/
cifar10/testing.csv
```

HEP-BNN is **launched** by running the following command in the root folder (*'fastinference/'*):

```
fastinference/implementations/neuralnet/
cuda/automatic/test_cuda.py
--outpath tmp/fastinference/cuda_auto
--dataset cifar
```

TABLE I
STRUCTURE OF THE CIFAR-10 BNN MODELS.

CIFAR-10 BNN model Structure
In → C64 → S → C64 → MP16 → S → C256 → S → C256 → MP8 → S → C512 → S → C512 → MP4 → S → FLAT → FC1024 → S → FC1024 → 10

TABLE II
STRUCTURE OF THE FASHIONMNIST BNN MODELS.

FashionMNIST BNN model structure
In → C64 → MP14 → S → C64 → MP7 → S → FLAT → FC2048 → S → FC2048 → 10

E. Generated files

In this section, we present the list of **generated files** for each configuration, including a brief description.

- *'utils.h'* and *'utils.cuh'*: contain utility functions (for C++ and CUDA code respectively)
- *'cuda_kernel.h'* and *'cuda_model.h'*: are headers containing functions that link the C++ code to the CUDA code
- *'modelW.hpp'*: contains declarations of the output arrays for every layer, and stores the weights, biases, and thresholds
- *'model.h'* and *'model.cpp'*: depending on the configuration, contains either the sequential C++ model for the CPU, or the calls to parallel CUDA model for the GPU (both implementations are present, but the unused part is commented out for debugging and comparison purposes)
- *'model.cu'*: CUDA code for the GPU (if applicable)

There are also a few files which are the same, regardless of configuration, which are already written and **copied** from the *'../cuda/automatic/'* folder to every generated implementation:

- *'main.cpp'*: splits the dataset into batches, calls the model predictor and calculates the accuracy and latency
- *'CMakeLists.txt'*: generates the 'Makefile' that compiles the entire project

Note that file changes to any of the *Jinja2* templates and 'implement.py' files require the following command to be run *'python setup.py install'*, before calling the *'make'* command.

IV. EXPERIMENT SETUPS AND RESULTS / EVALUATION

In this section, we present our experimental results. Information about the hardware, datasets, and models are presented in Section IV-A. In Section IV-B, the results of our *HEP-BNN* framework, i.e., implementing the suitable configurations for every layer, are presented and compared to the baseline sequential implementation, as well as other parallel configurations.

A. Profiling Environment

1) ***Hardware****:* We execute our experiments on a Linux based server equipped with an Intel Core i7-8700K CPU and a GTX1080 GPU with 8GB of video memory each. Additionally, we run the same experiments on a Windows-system with a consumer-grade GPU (GTX 1650Ti), as well as on an embedded Jetson TX2 board. Table III also lists the amount of available CUDA cores for each GPU.

TABLE III
OVERVIEW OF HARDWARE USED FOR EVALUATION

Name	CPU	GPU	CUDA Cores
Server	i7-8700K	GTX1080	2560
Laptop	i7-10750H	GTX1650Ti	1024
Jetson TX2	Cortex-A57	Pascal-based	256

Kernel launches are wrapped around `cudaEventRecord()` functions, in order to accurately measure the GPU-time. The communication cost (i.e. memory allocation and transfer between host and device) in the CUDA code is executed before the kernel launch, by the CPU, and is included in the CPU-overhead time. We implement independent layers for the models, therefore data transfer between CPU and GPU takes place before and after every layers execution (even if two consecutive layers are executed on the GPU). The code generator can be adapted to consider this case in future works/implementations.

2) ***Datasets****:* The algorithm is evaluated on two commonly used benchmarking datasets, namely FashionMNIST and CIFAR-10. The FashionMNIST [9] dataset consists of $70,000$ gray-scale images and labels from 10 classes, representing different clothing articles. The size of each image is 28×28 pixels in 1 channel, with 0 representing the *brightest* and 255 the *darkest* values. Out of the total amount of images, $60,000$ are used for training, while the remaining $10,000$ for testing. The CIFAR-10 [10] dataset contains $60,000$ colour images (3 channels), each with a size of 32×32 for a total of 1024 pixels. It is split into $50,000$ training and $10,000$ test images, which are classified in 10 different classes representing means of transportation (i.e. airplane, ship, truck, automobile) and animals (i.e. bird, cat, dog, deer, frog, horse).

3) ***BNN Architecture****:* The BNN models used for inferring the datasets are VGG-type architectures [11] adapted for the binarized variant of NNs. The FashionMNIST network model contains a total of 10 layers, each of them belonging to one of the types presented in Section II-A. Specifically, the 1^{st} and 4^{th} layer are convolutional layers, with a size of $28 \times 28 \times 64$ and $14 \times 14 \times 64$ respectively. The convolutional layers are down-sampled to half of their input size, in the immediately following maxpool layers, namely in the 2^{nd} and 5^{th} layer. Step layers are employed on the 3^{rd}, 6^{th}, and 9^{th} layer, which apply batch normalization and the activation function. After convoluting and down-sampling the input image, it is flattened into a 1-dimensional array in layer 7. Finally, a total of 2048 neurons are fully-connected in the 8^{th} and 10^{th} layer.

The structures of the networks are listed in Table II and I, using the notations from Section II-A. The CIFAR-10 model also contains the standard layer types for BNNs, totalling to 19 layers. Convolutional layers are placed on the 1^{st}, 3^{rd}, 6^{th}, 8^{th}, 11^{th}, and 13^{th} position. Down-sampling using maxpooling occurs only three times, namely in the 4^{th}, 9^{th}, and 14^{th} layer. The 16^{th} layer flattens the image for the fully-connected layers at position 17 and 19. The rest of the layers are step layers.

To simulate the inference process, the sets of test images

TABLE IV
EFFICIENT CONFIGURATIONS FOR THE CIFAR10 MODEL

	C64	S	C64	MP16	S	C256	S	C256	MP8	S	C512	S	C512	MP4	S	FLAT	FC1024	S	FC1024
Server	**CPU**	CPU	**Z**	**CPU**	CPU	**XZ**	CPU	**XYZ**	**XY**	CPU	**XZ**	CPU	**XZ**	**X**	CPU	CPU	**X**	CPU	**CPU**
Laptop	**CPU**	CPU	**Y**	**CPU**	CPU	**XYZ**	CPU	**XYZ**	**CPU**	CPU	**XYZ**	CPU	**XYZ**	**CPU**	CPU	CPU	**X**	CPU	**CPU**
TX2	**CPU**	CPU	**XYZ**	**CPU**	CPU	**Z**	CPU	**Z**	**CPU**	CPU	**XZ**	CPU	**XZ**	**CPU**	CPU	CPU	**XY**	CPU	**CPU**

TABLE V
EFFICIENT CONFIGURATIONS FOR FASHION-MNIST MODEL

	C64	MP14	S	C64	MP7	S	FLAT	FC2048	S	FC2048
Server	**CPU**	**CPU**	CPU	**XZ**	**X**	CPU	CPU	**CPU**	CPU	**CPU**
Laptop	**CPU**	**CPU**	CPU	**CPU**	**CPU**	CPU	CPU	**CPU**	CPU	**CPU**
TX2	**CPU**	**CPU**	CPU	**XZ**	**CPU**	CPU	CPU	**CPU**	CPU	**CPU**

from both models respectively are used. This guarantees a controlled benchmarking environment for the algorithm that profiles each layer of the BNN models. The weights and biases were trained over the course of 100 epochs, and achieve an inference accuracy of 77.24% for the FashionMNIST, and 67.08% for the CIFAR-10 model.

Preliminary observations showed that the batch size has an impact on runtime for certain configurations. Therefore, the experiments also apply different batch sizes for the two BNN models, from $\{1, 2, ..., 128\}$, in increments of powers of 2.

B. Results

Table VI presents the inference times measured by running the BNN models (each with 10000 data images as input), for every tested target hardware. The latency (displayed in seconds) is the time required by the BNN to process the **entire** test dataset of 10000 images. Recall from Section III-B, that the suitable configuration is chosen over all the different batch sizes, and therefore it is important to note the batch size alongside the runtime. Specifically, a batch size of n means that n data-images are processed in parallel.

It can be observed that the server, which has the most CUDA cores out of the three tested hardware, has overall faster runtimes. On the other hand, the resource-constrained TX2, having the least amount of CUDA cores, has significantly higher runtimes.

The suitable configurations mapped for each layer is presented in Tables IV and V for the *CIFAR-10* and *FashionMNIST* models respectively. From there, we can notice the following observations:

- Since the Fashion-MNIST BNN model is smaller out of the two, almost all of the layers are mapped to the CPU for sequential execution. A notable exception is the second convolution layer, which is mapped to the *XZ* configuration on the Server and the TX2.
- In the case of the CIFAR10 BNN model, we observe that the maxpool layer is mapped to the CPU in almost every case, whereas the convolutional and fully-connected layers are mapped to the GPU using different parallel configurations. For example, the second convolutional layer is mapped to the *Z* configuration on the server, *Y* on the Laptop and *XYZ* on the TX2.

Finally, Figure 5 compares the purely sequential CPU implementation and two intuitive ways of parallelizing the BNN models to the results of the mapping algorithm. The naive GPU implementation considers the parallelization of every layer suitable for GPU acceleration using only the *Data (X)* configuration. The full-parallel GPU implementation parallelizes every suitable layer as much as possible, i.e., applying the *Data + Window + Neuron (XYZ)* configuration. These are the two most intuitive ways of parallelizing the workload, while not considering the fact that some layers may not benefit from GPU acceleration.

TABLE VI
THE MINIMUM INFERENCE TIMES OF THE EFFICIENT CONFIGURATIONS

Dataset	Fashion-MNIST			CIFAR10		
Hardware	Server	Laptop	TX2	Server	Laptop	TX2
Runtime	2.11s	2.84s	9.31s	41.6s	55s	297s
Batch size	16	2	64	16	128	16

The results from running the efficient configurations for every batch size, show a significant speedup overall. Specifically, compared to the fully-parallel implementation, running the *HEP-BNN* framework on the server leads to at most $2\times$ speedup, while on the Jetson TX2, it achieves at most $2.6\times$ speedup, and on the Laptop-system the efficient configuration results in a $11.8\times$ improvement.

V. CONCLUSION

We propose a framework that generates efficient BNN layer-to-device mappings for heterogeneous multiprocessor platforms comprised of CPU and CUDA-capable GPU. Given a trained BNN model, our proposed *HEP-BNN* framework systematically evaluates the execution time of the model on CPU and on GPU under different parallel configurations. We evaluate our framework with two BNN architectures on well-known datasets, running on three different types of hardware platforms. The results show that, across the tested dataset-s/BNNs and the different hardware platforms, our proposed framework generates mappings for BNN inference which achieve significantly higher speedup compared to a fully-parallelized approach. Specifically, the efficient parallel configuration from our *HEP-BNN* framework reduces inference times by up to $2\times$, $2.6\times$ and $11.8\times$, across the tested target hardware respectively. The generated GPU code from *HEP-BNN* containing the efficient configuration can also then be used for applications using BNN inference in practice.

We believe that our *HEP-BNN* framework will benefit researchers and practitioners to find efficient execution configurations for BNN inference systems using heterogeneous platforms comprised of CPU and GPU.

ACKNOWLEDGMENT

This paper has been supported by Deutsche Forschungsgemeinschaft (DFG) project OneMemory (405422836), by the Collaborative Research Center SFB 876 "Providing Information by Resource-Constrained Analysis" (project number 124020371), subproject A1 (http://sfb876.tu-dortmund.de) and by the Federal Ministry of Education and Research of Germany and the state of NRW as part of the Lamarr-Institute for

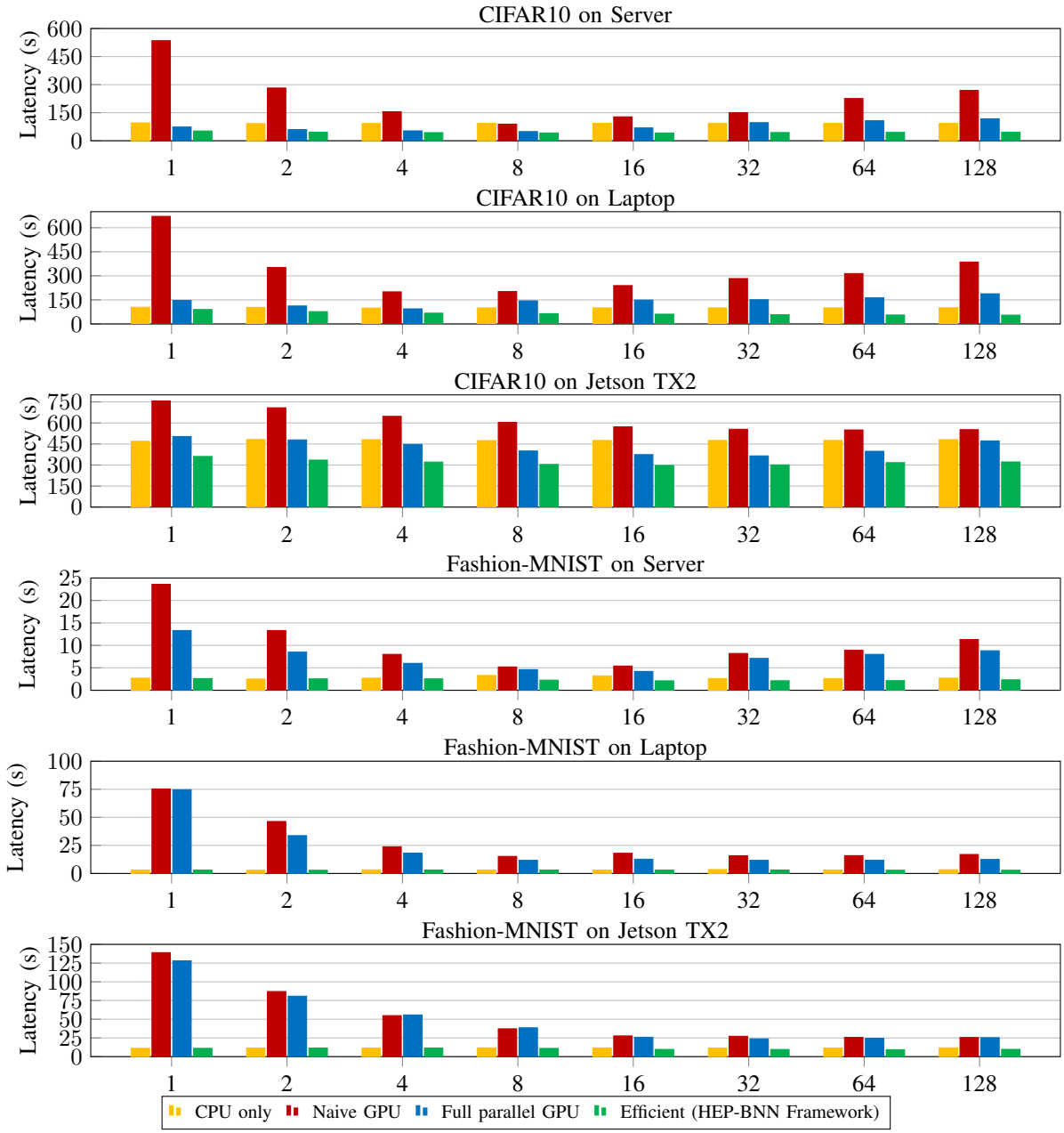

Fig. 5. Execution time over batch size comparison for the entire FashionMNIST test images dataset (upper three figures) and entire CIFAR10 test images dataset (lower three figures). Each dataset is evalauted with three different hardware configurations, namely: Server, Laptop, and TX2.

ML and AI, LAMARR22B. This work has received funding by the German Federal Ministry of Education and Research (BMBF) in the course of the 6GEM research hub under grant number 16KISK038.

References

[1] O. I. Abiodun, A. Jantan, A. E. Omolara, K. V. Dada, N. A. Mohamed, and H. Arshad, "State-of-the-art in artificial neural network applications: A survey," *Heliyon*, vol. 4, no. 11, p. e00938, 2018.

[2] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks," in *Advances in Neural Information Processing Systems (NIPS)*, 2016.

[3] E. Nurvitadhi, D. Sheffield, J. Sim, A. Mishra, G. Venkatesh, and D. Marr, "Accelerating binarized neural networks: Comparison of fpga, cpu, gpu, and asic," in *2016 International Conference on Field-Programmable Technology (FPT)*, pp. 77–84, 2016.

[4] X. Xu and M. Pedersoli, "A computing kernel for network binarization on pytorch," *CoRR*, vol. abs/1911.04477, 2019.

[5] A. Li and S. Su, "Accelerating binarized neural networks via bit-tensor-cores in turing gpus," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 7, pp. 1878–1891, 2021.

[6] G. Chen, S. He, H. Meng, and K. Huang, "Phonebit: Efficient gpu-accelerated binary neural network inference engine for mobile phones," in *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 786–791, 2020.

[7] E. Sari, M. Belbahri, and V. P. Nia, "How does batch normalization help binary training?," *arXiv:1909.09139*, 2019.

[8] S. Buschjäger, "fastinference github repository." https://github.com/sbuschjaeger/fastinference.

[9] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms," 2017.

[10] A. Krizhevsky, "Learning multiple layers of features from tiny images," tech. rep., 2009.

[11] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *International Conference on Learning Representations, (ICLR)*, 2015.