This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/LES.2021.3087707, IEEE Embedded Systems Letters

Deep Learning Inference Parallelization on Heterogeneous Processors with TensorRT

EunJin Jeong*, Jangryul Kim*, Samnieng Tan, Jaeseong Lee, and Soonhoi Ha, *Fellow, IEEE*

***Abstract*—As deep learning inference applications are increasing, an embedded device tends to equip neural processing units (NPUs) in addition to a CPU and a GPU. For fast and efficient development of deep learning applications, TensorRT is provided as the SDK for NVIDIA hardware platform, including optimizer and runtime that delivers low latency and high-throughput for deep learning inference. Like most deep learning frameworks, TensorRT assumes that the inference is executed on a single processing element, GPU or NPU, not both. In this paper, we propose a parallelization methodology to maximize the throughput of a single deep learning application using both GPU and NPU by exploiting various types of parallelism on TensorRT. With six real-life benchmarks, we could achieve 81% ~ 391% throughput improvement over the baseline inference using GPU only.**

***Index Terms*—Deep Learning, Optimization, Acceleration**

I. Introduction

As deep learning (DL) inference applications are increasing in embedded devices, an embedded device tends to equip hardware accelerators in addition to a multi-core CPU and a GPU. As an example, the NVIDIA Jetson AGX Xavier board (Xavier) contains a CPU, GPU, and deep learning accelerators (DLAs), which is the assumed hardware platform in this paper. A DLA is also called a neural processing unit (NPU). To run a trained network on an embedded device, we usually use the software development kit (SDK) provided with the device. For the fast and efficient development of deep learning applications, TensorRT [1] is provided as the SDK for an NVIDIA device.

Extensive researches have been conducted to accelerate DL applications via software optimization such as quantization, low-rank approximation, and layer fusion. Some optimization techniques are adopted in TensorRT. In this work, we aim to maximize the throughput of a DL application by exploiting various levels of parallelism including task-level parallelism and pipelining, which we call *network-level* optimization.

Network-level optimization techniques depend on the hardware platform and the objective function. Tang et al. [2] aim to maximize resource utilization on a multi-core CPU while a framework proposed in [3] aims to improve the throughput on heterogeneous CPU cores. DeepX [4] and CNNdroid [5] use a GPU to accelerate a single execution of a DL application on a mobile device. A throughput maximization technique by CPU/GPU pipelining was presented in [6], in which pre/post-processing (*Pre./Post.*) tasks are run on a CPU while the network body is run on a GPU in a pipelined fashion. The authors of [7] consider multiple deep neural networks (DNNs) running on a CPU-GPU heterogeneous system. They focus on satisfying the response time guarantee of high priority DNNs while maximizing the throughput of low priority DNNs. Since they deal with multiple networks, they aim to increase the overall throughput of multiple DNNs by pipelining or batch processing, assuming that each application is run serially. Instead, we aim to maximize the throughput of a single application by overlapping multiple iterations. While they make a pipelining decision based on the profiled execution time of layers, we make a sub-optimal decision without profiling data since layer-wise profiling is not available or inaccurate at best in our target system. The work of [8] converts a CNN model to a dataflow graph and uses a conventional parallel scheduling technique of a dataflow graph onto a multiprocessor system.

There exist a few recent studies that consider NPU as well as a GPU for DNN acceleration. The authors in [9] presented how to schedule multiple DNN instances with real-time constraints on the Xavier, assuming each network is mapped onto a single processing element (PE). Another approach proposed in [10] also considers multiple DNN instances on a heterogeneous system that includes both GPU and NPU, exploiting task-level parallelism of each network. While maximizing the throughput is included as an objective function, NPUs are not included in the experimental results since there is no SDK that supports both GPU and NPU for a single DL application.

In this paper, we present a TensorRT-based parallelization method that uses both GPU and NPUs to maximize the throughput of a single DNN application. First, multiple threads are used to parallelize *Pre./Post.* steps of inference. Also, accelerators are fully exploited by using multiple streams. Second, we propose a heuristic to explore the wide design space of pipelining the network. The design space is defined by several design parameters such as the number of pipeline stages, pipeline cut-points, and the number of streams. Lastly, we duplicate a part of the network and maps them onto different DLAs while sharing the GPU for the remaining part. By running two network instances concurrently, we may improve the utilization of heterogeneous processors.

With six real-life benchmarks, we could achieve 81% ~ 391% performance improvement over the baseline inference that is performed on a GPU. It proves the significance of parallelization by pipelining and multi-threading in an embedded device. To the best of our knowledge, there is no study of parallelization on the TensorRT SDK with GPU and DLAs together. Although the proposed technique is verified on the specific hardware platform, we believe that the proposed methodology of DSE could be applicable to others.

II. Background

A. NVIDIA Jetson AGX Xavier

In this work, we used an NVIDIA Jetson AGX Xavier as an example embedded device with heterogeneous accelerators to evaluate the proposed technique and methodology. The board

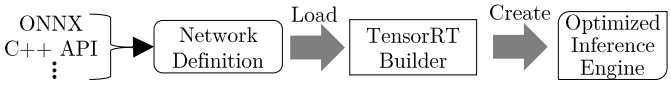
*: Both authors contributed equally to this work.

The authors are with the Department of Computer Science and Engineering, Seoul National University, Seoul, Republic of Korea. (e-mail: {chjej202, urmydata}@snu.ac.kr, tansamnieng@iris.snu.ac.kr, {tbvj5914, sha}@snu.ac.kr)

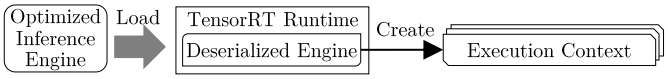
Corresponding author: Soonhoi Ha.

This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (No. NRF-2019R1A2B5B02069406).

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/LES.2021.3087707, IEEE Embedded Systems Letters



(a) The workflow of generating inference engine in TensorRT



(b) The workflow of generating execution context in TensorRT

Figure 1: Overall workflow of NVIDIA TensorRT

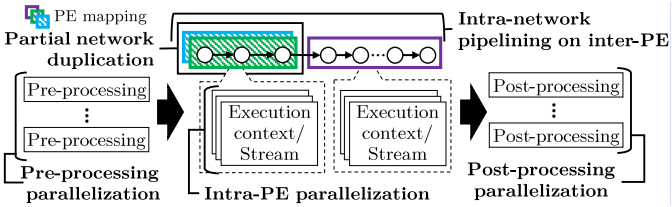


Figure 2: Four parallelization techniques

contains an octa-core ARMv8 CPU, one Volta GPU, and two NVIDIA DLAs. The DLA is a power-efficient accelerator, but its computational power is weaker than GPU. The Xavier has a unified memory shared by CPU and GPU. It means that communication between CPU and GPU can be easily conducted by memory operation with the potential risk of access contention. Communication between a DLA and another processing element is included in TensorRT APIs.

B. NVIDIA TensorRT

TensorRT [1] is an SDK for fast inference. Figure 1 (a) shows the workflow of TensorRT that creates an optimized inference engine by the *Builder* module with network definition. In this step, TensorRT internally applies some optimization techniques such as layer fusion. The optimized engine creates the execution context, which performs the inference, as shown in Fig. 1 (b). Each execution context corresponds to a CUDA stream, so multiple streams can be executed concurrently inside a GPU or a DLA. The *Runtime* module loads the serialized engine and maps a GPU or NPU to the engine. Note that it is possible to build multiple contexts from the same engine and map them to distinct streams to perform inference concurrently. TensorRT assumes that the inference is executed on a single processing element, GPU or NPU, not both.

III. Parallelization Methodology

We aim to improve the throughput performance of a single CNN inference application by utilizing all available processing elements. The proposed methodology consists of four main techniques, as outlined in Fig. 2.

A. Pre/Post-Processing Pipelining and Parallelization

Pre- and post-processing are essential parts to run DL applications. In the pre-processing step, we load an input image and re-size the image. After the completion of inference, we perform post-processing for localizing detected objects and storing results. Pipelining the *Pre./Post.* part with the main inference body is popularly used to improve the performance [6]. However, just pipelining those parts is not enough in some cases. *Yolov4-tiny* network, as an example, it is necessary to utilize more than two *Pre.* threads concurrently since it may take longer to *Pre.* than the time for inference body. Thus, we parallelize the *Pre./Post.* part with multiple threads in case the part becomes the performance bottleneck.

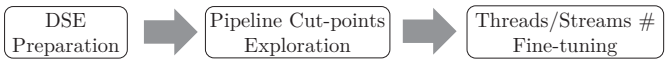


Figure 3: The workflow of the proposed DSE methodology

B. Intra-PE Parallelization

A data-parallel accelerator such as GPU can parallelize multiple instances of the assigned kernel using multiple streams, which increases the utilization further. Independently of the number of threads in the *Pre./Post.* steps, we can create more than one stream in GPU and DLA by creating as many buffers as the number of streams at the pipeline-interface. It is observed that if the number of streams exceeds a certain level, the performance is saturated. Thus we set the number of streams as an optimization parameter.

C. Intra-network Pipelining

To use all available accelerators, we pipeline the inference body. It is necessary to determine how to split the network into stages and how to assign the stages to the PEs. Since the design space of pipelining is huge, how to explore the space is a challenging problem. To tackle this challenge, we devise a heuristic to decide the cut-points for network splitting for a given mapping option which is explained in the next section.

D. Partial Network Duplication

Lastly, we may duplicate the part of the network, which is called partial network duplication (PND). It is important to balance the execution time among the pipeline stages. Since it is easier to balance one GPU and one DLA than one GPU and two DLAs, we pipeline the network onto one GPU and one DLA in this technique. Then we run two iterations of the network concurrently, mapping the duplicated part of the network to different DLAs and sharing the GPU for the remaining part. In Fig. 2, the striped boxes are mapped onto different DLA with the separate inference engine, and each engine is responsible for half of the streams in the stage.

IV. Design Space Exploration

Based on the techniques introduced in Section III, the design space to explore for throughput optimization is defined by the following parameters: (1) The number of *Pre./Post.* threads, (2) The number of streams, (3) pipelining cut-points on the given mapping, (4) whether applying partial network duplication or not. Figure 3 shows the workflow of the proposed exploration methodology of the design space.

A. DSE Preparation and Fine-tuning

First, we find the upper bound on the number of streams through preliminary experiments using the GPU only. The number of streams is increased until the throughput stops improving. For easy buffer management, we set the number of streams as a multiple of the number of threads in the pre/post-processing steps. With the maximum number of streams, we explore the design space of pipelining with the proposed heuristic. After pipelining decision is made, we reduce the number of streams for fine-tuning in the last step as long as the performance is not degraded to minimize the resource usage.

B. Pipeline Cut-points Exploration

In this step, we explore cut-points on the given mapping. Since our target board has one GPU and two DLAs, the mapping space is limited. We define four feasible pipelining options as

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/LES.2021.3087707, IEEE Embedded Systems Letters

Algorithm 1 The network pipelining heuristic

Require: $cut_{init/cur/prev}$: an initial/current/previous cut-point tuple
Require: T: Threshold value of GPU utilization
Require: $fps_{cur/prev}$: FPS of the run with $cut_{cur/prev}$
Require: $gpuUtil_{cur}$: GPU utilization with current tuple of cut-points
Require: $TopK$: the list of top 10 cut-point tuples in FPS
Require: P: The set of move policies to explore the cut-point tuples.
1: $cut_{cur} = cut_{init}$, cut_{prev} =None, $fps_{prev} = MAX$
2: fps_{cur}, $gpuUtil_{cur}$ = Run program with cut_{cur}
3: **while** New cut-point tuple is selected **do**
4: Select P with $gpuUtil_{cur}$
5: **for** each P_i in P **do**
6: $cut_{prev} = cut_{cur}, fps_{prev} = fps_{cur}$
7: cut_{cur} = Select a new cut-point tuple with (cut_{cur}, P_i)
8: fps_{cur}, $gpuUtil_{cur}$ = Run program with cut_{cur}
9: Update $TopK$ with (cut_{cur}, fps_{cur})
10: **if** $gpuUtil_{cur} \geq T$ and $cut_{cur} \in TopK$ **then**
11: Perform local search with cut_{cur}
12: **end if**
13: **if** $fps_{prev} < fps_{cur}$ **then**
14: continue with P_i again
15: **end if**
16: **end for**
17: **end while**

shown in Table I. Because DLA only supports the limited types of layers, some layers are not runnable on a DLA. Since such layers exist in the bottom part of the benchmark networks used in the experiments, we assign the last pipeline stage to GPU in all options. In options *C* and *D*, we use two DLAs. In options *B* and *D*, GPU is assigned two pipeline stages since it has bigger computation power than DLA. For options *A* and *B* that use a single DLA, we can use the partial network duplication technique. Thus, there are six different mapping options to be compared with each other to find the best mapping for a given network. For each option, we find a sub-optimal tuple of cut-points for each pipelining option with the proposed heuristic.

To avoid the exponential complexity of exhaustive search, a 2-phase heuristic is devised to find a sub-optimal tuple of cut-points. The first phase is a global search over a sampled set of cut-points: we prune the search space by sampling the cut-points regularly. Algorithm 1 shows the first phase of the proposed heuristic with the sampled cut-points. The other cut-points are explored in the second phase, *local search*.

First, an initial cut-points tuple, cut_{init}, is determined based on the profiling information of layer execution time on processing elements, GPU and NPU. We obtain the FPS and GPU utilization by running the program with the initial cut-points (line 2). We expand the searching area of cut-points by defining several cut-moving policies from the current tuple of cut-points. If the GPU utilization, $gpuUtil_{cur}$ is lower than a given threshold T, we move the cut-points to the direction of increasing the GPU utilization by adopting the binary search method for fast search. Otherwise, we search the cut-points around the current cut-points by using the following two policies; One is to move a single cut-point by one and the other is to shift all cut-points in the left or right direction. We use all policies to explore diverse candidate pipelining solutions around the current cut-point tuple for all networks (line 5). After the policy set is selected based on $gpuUtil_{cur}$ (line 4), we obtain a new candidate tuple of cut-points according to each moving policy P_i (line 7). After we measure the FPS and GPU utilization by running the program (line 8), we update the list of cut-point tuples with top 10 FPS performance, $TopK$.

If the GPU utilization, $gpuUtil_{cur}$, becomes no smaller than T and the current tuple, cut_{cur}, is newly included in $TopK$, the second phase, local search, is performed (lines 10-12). In the local search phase, each cut-point in cut_{cur} is moved one by one to explore the cut-points that do not belong to the sampled cut-points in the first phase. If fps_{cur} is higher than fps_{prev}, the current moving policy is maintained by using the same policy P_i at the next iteration (line 13-15) rather than moving to P_{i+1}. The overall heuristic is performed until no new cut-point tuple is selected by all moving polices.

Table I: Options for intra-network pipelining

Option	# of pipeline stages	Composition of PEs
A	2	DLA - GPU
B	3	GPU - DLA - GPU
C	3	DLA - DLA - GPU
D	4	GPU - DLA - DLA - GPU

Note that the performance after cut-point moving cannot be estimated analytically due to the unknown effect of software optimization of TensorRT. Also, the layer-wise execution time cannot be obtained from the DLA in the Xavier board. Thus, we run the program on the target board for each searched tuple of cut-points to measure the FPS and GPU utilization. Note that communication time between pipelining stages is naturally considered in the heuristic. The run-time of the proposed heuristic depends on the number of searched cut-point tuples. In addition, we need to build and store the optimized engine for each pipelining stage based on the mapping and cut-points information. Hence it was necessary to automate this process and reuse the pre-built engines as much as possible.

V. Experiments

A. Setup

All experiments were conducted on a Jetson AGX Xavier board with Jetpack 4.3 and TensorRT 6. We used the tkDNN [11] library that makes TensorRT easy to use. Since it does not support pipelining, however, we modified the library to create a separate inference engine for each pipeline stage. Table II lists the benchmark networks supported by tkDNN; They are all object detection networks. Since the DLA does not support *leaky relu* or *mish* activation currently, we replaced those with *relu* activation and retrained the networks. If there is any layer that the DLA does not support among the layers mapped to the DLA, the layer is actually executed on the GPU, which is called *GPU fallback*. As for the dataset, we use the *COCO2014 trainval* for training and the *COCO2017 val* for inference with image size 416x416. We set the TensorRT batch size to 1 for each stream except for the batch processing experiment that will be explained below. We set the maximum frequency on the *MAXN* power mode, which does not limit the power budget. In the pipelining heuristic, we use 5,000 test images to estimate the FPS of each candidate set of cut-points. After the final configuration is determined, we perform each experiment five times and get the average value.

B. Design Space Exploration Results

We compare six options after the proposed methodology is applied, and Fig. 4 shows the comparison results. We calculate the relative FPS ratio of each option to the lowest FPS for each network. For all networks except for *Yolov4-tiny*, options *C* and *D* give higher FPS than options *A* and *B* without PND since it uses one more DLA. In *Yolov4-tiny*, some layers in the front of the network are not runnable on DLA, so option *C* incurs GPU fallback. Note that using PND is always beneficial

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/LES.2021.3087707, IEEE Embedded Systems Letters

Table II: Configurations of selected options on each experiment

Network	Label	# of layers	Profiling			Heuristic without partial network duplication			Heuristic with partial network duplication		
			Opt.	Cuts.	(Pre. Post. Str.)	Opt.	Cuts.	(Pre. Post. Str.)	Opt.	Cuts.	(Pre. Post. Str.)
Yolov2 [12]	Y2	54	D	19/27/38	(2, 1, 8)	C	5/22	(2, 1, 8)	A	19	(2, 1, 8)
Yolov3 [13]	Y3	179	C	17/54	(1, 1, 4)	C	20/57	(1, 1, 4)	A	57	(1, 1, 4)
Yolov4 [14]	Y4	269	D	42/90/142	(1, 1, 4)	D	37/77/134	(1, 1, 4)	B	42/131	(1, 1, 4)
Yolov4-tiny [14]	Y4t	57	D	15/26/32	(4, 1, 8)	D	18/27/30	(4, 1, 8)	B	18/30	(4, 1, 8)
CSPNet [15]	CN	228	A	16	(1, 1, 4)	C	21/34	(1, 1, 4)	B	2/31	(1, 1, 4)
Densenet+Yolo [16]	DY	508	D	80/203/298	(1, 1, 4)	D	75/203/298	(1, 1, 4)	A	97	(1, 1, 4)

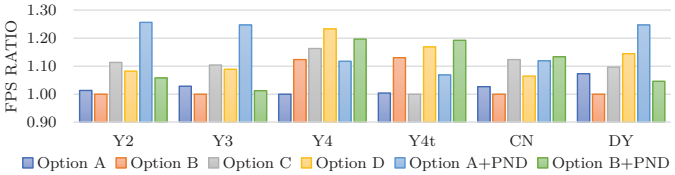


Figure 4: FPS comparison among options

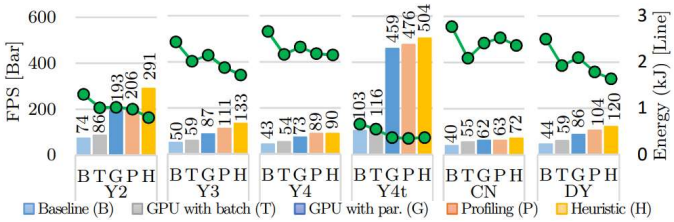


Figure 5: FPS and energy comparison among five methods

in options *A* and *B*, and there is no single best option for all networks. Since GPU is more powerful than DLAs, more layers are assigned to GPU in all benchmarks. For CSPNet that uses grouped convolution in early layers, such unbalance between the number of layers mapped to GPU and a DLA is apparent. Table II shows the detailed information on the best parallelization configuration for each network and each method in which *Opt.* is the mapping option in Table I, and *Cuts.* indicates the cut-point tuple. *(Pre,Post,Str)* indicates the number of *Pre./Post.* threads and streams, respectively.

The total running time of the proposed technique depends on the number of layers and the convergence speed to a local minimum in the heuristic. The worst-case running time was about 3.5 days for the Densenet+Yolo benchmark, and the number of cut-point tuples explored was about 1,000 which covers only 0.0045% of the whole design space.

C. Comparison with the Other Methods

We compare the proposed technique with the other four methods, as shown in Fig. 5. The baseline is the default TensorRT implementation where a single execution context is mapped to a GPU stream. The second method is batch processing by setting the TensorRT batch size to 32 and using OpenMP for pre-/post-processing steps. In this method, DLAs are not used, and no CPU-GPU pipelining is applied. The third method applies the proposed parallelization technique to the system without using DLAs. The fourth method is to balance the pipeline stages as much as possible, based on the profiled information only without the proposed pipelining heuristic. The execution time of each layer on GPU could be obtained by using the TensorRT IProfiler. Since per-layer profiling is not available for DLA, we estimate the execution time on a DLA by weighting the GPU execution by the average performance ratio between DLA and GPU based on the total inference time.

Figure 5 presents the FPS and energy consumption of five methods. By applying the parallelization technique on a single GPU, the FPS is increased significantly, 56% to 347%, compared to the baseline. Profiling-based pipelining shows better FPS than the GPU-only method in all benchmarks. It confirms the motivation of this work that aims to utilize all available processing elements for throughput optimization. In all cases, the proposed technique gives the best throughput. In particular, the proposed technique improves the throughput by up to 41%, compared with the profiling-based method. In terms of energy consumption, it is observed that our heuristic method saves energy up to 20% compared to the third one. Although the heuristic method consumes more power because of using two DLAs, the energy consumption is reduced since the overall execution time is shortened and a DLA is more power-efficient than the GPU. For the CSPNet benchmark, the batch processing method gives the lowest energy consumption, which shows the trade-off between the FPS performance and the energy consumption.

VI. Conclusion

In this paper, we present a parallelization methodology to accelerate a single application by exploiting various types of parallelism: multi-threading, multi-stream, pipelining of the inference network, and partial network duplication. To explore the wide design space of pipelining, we devise a heuristic to determine the pipeline cut-points. The proposed methodology is evaluated with six real-life object detection networks on an NVIDIA Jetson AGX Xavier board. We could achieve 81% ~ 391% throughput improvement over the baseline inference that uses the GPU only.

References

[1] "NVIDIA TensorRT," https://developer.nvidia.com/tensorrt/.
[2] L. Tang *et al.*, "Scheduling computation graphs of deep learning models on manycore cpus," *arXiv*, 2018.
[3] S. Wang *et al.*, "High-throughput cnn inference on embedded arm big.little multi-core processors," *IEEE TCAD*, 2019.
[4] N. D. Lane *et al.*, "DeepX: A software accelerator for low-power deep learning inference on mobile devices," in *IPSN*, 2016.
[5] S. S. L. Oskouei *et al.*, "CNNdroid: GPU-accelerated execution of trained deep convolutional neural networks on android," in *MM*, 2016.
[6] D. Kang *et al.*, "Joint optimization of speed, accuracy, and energy for embedded image recognition systems," in *DATE*, 2018.
[7] Y. Xiang *et al.*, "Pipelined data-parallel cpu/gpu scheduling for multi-dnn real-time inference," in *RTSS*, 2019.
[8] S. Minakova *et al.*, "Combining task-and data-level parallelism for high-throughput cnn inference on embedded cpus-gpus mpsocs," in *IC-SAMOS*, 2020.
[9] R. Pujol *et al.*, "Generating and exploiting deep learning variants to increase heterogeneous resource utilization in the nvidia xavier," in *ECRTS*, 2019.
[10] D. Kang *et al.*, "Scheduling of deep learning applications onto heterogeneous processors in an embedded device," *IEEE Access*, 2020.
[11] M. Verucchi *et al.*, "A systematic assessment of embedded neural networks for object detection," *ETFA*, 2020.
[12] J. Redmon *et al.*, "Yolo9000: better, faster, stronger," in *CVPR*, 2017.
[13] J. Redmon and A. Farhad, "Yolov3: An incremental improvement," *arXiv*, 2018.
[14] A. Bochkovskiy *et al.*, "Yolov4: Optimal speed and accuracy of object detection," *arXiv*, 2020.
[15] C.-Y. Wang *et al.*, "CSPNet: A new backbone that can enhance learning capability of CNN," in *CVPR Workshop*, 2020.
[16] "Densenet201+Yolo," https://github.com/AlexeyAB/darknet/.